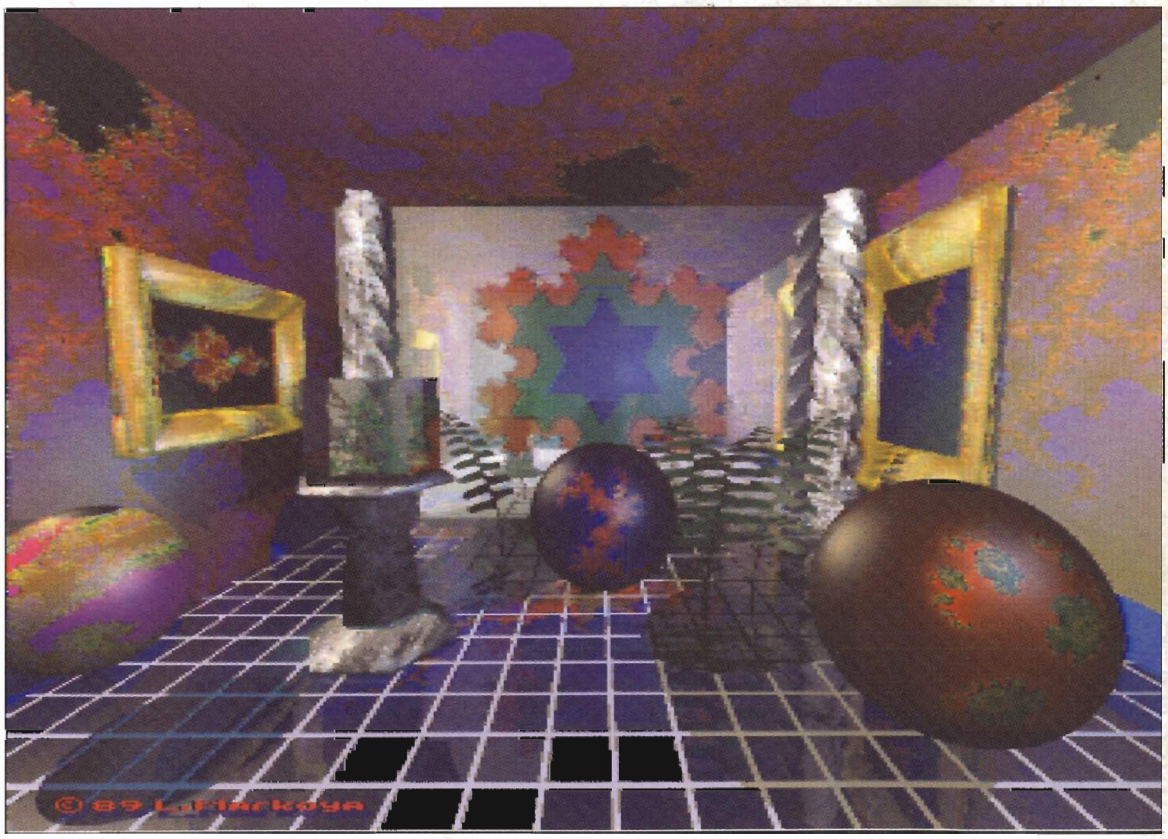
JUNE 1989: VOLUME 2, ISSUE 3

THE MAGAZINE FOR AMIGA PROGRAMMERS

Canada $4.25
USA $3.50

FOR THE Amiga Transactor

- First steps in C++ *by Richie Bielak*
- Getting a handle on file systems, part 2 *by John Toebes & Doug Walker*
- Animation: Keep it simple! *by Derek Grime*
- Assembly language programming *by Jim Butterfield*
- Dynamics for animators *by Eric and Cathryn Graham*
- Colour video hardware design *by Ben Blish*
- Interfacing to assembly language in Modula-2 *by Anthony Bryant*
- What characters! AmigaDOS wildcards in Modula-2 *by Clark Williams*
- **Reviews:** Lattice C v5.02, CEDPro text editor, Carl Sassenrath on interrupts, flickerFixer, LaTeX, *UNIX: The Complete Reference*
- **Plus** regular columns by Larry Phillips, Don Curtis and Steve Ahlstrom



The Fractal Gallery by Louis Markoya

06
29074 12122
0

Lattice Tools & Libraries

We are the company that writes the language and the tools. So who could know more about the utilities you need to maximize productivity?

Lattice 5.0 Cross Compiler
All the power and facilities of the Lattice AmigaDOS 5.0 compiler in a cross-compiler MS-DOS and OS/2 host. Includes our full screen cross debugger. $750.

Lattice C++
Object Oriented C++ Programming for the Amiga. Provides object definitions for AmigaDOS, EXEC and Intuition. $300.

Lattice Compiler Companion
Collection of UNIX-like utilities that make your programming environment more productive. (Included in Lattice 5.0.) $100.

dBC III Library
Amiga programming library that provides programming interface into dBASE III compatible files. $150.

Communication Library
New Amiga programming library that supports modem interfaces for XMODEM, YMODEM, KERMIT and ASCII protocols. $250.

PANEL Library
Amiga programming library that supports windows graphics applications. $195.

C Programming Seminar
Learn C programming from Lattice experts. Four day seminar in Chicago. $895.

Call any of our world-wide distributors, or call us at 1-800-444-4309 or (312) 916-1600. Fax #(312) 916-1190, TELEX 532253.

"Brilliant!"

– *William Hawes* author of ARexx

"Lightning Fast!"

– *Reichart Von Wolfsheild* Silent Software, Inc.

"Blows away all the others."

– *Robert Berryhill* LMR Creations

Lattice AmigaDOS C 5.0 is getting lots of praise. And not just because of the latest benchmarks which prove Lattice C 5.0 is the best compiler by every measure.

For instance, Michael Berenstein calls 5.0 "Wonderful," while Roger Uzun simply calls this total programming environment "...the best."

Tom Dolan felt the same about CodePRobe, our new Source Level Debugger. Support for both C and Assembly language, plus multi-tasking debugging make it "...the best I have seen or used!!!!"

And our new Global Optimizer caused Steve Tibbett, a "VERY happy customer," to say, "WOW. This is outstanding!" and Tomas Rokicki of Radical Eye Software to exclaim "...Global Optimizer, inline subroutines and extensive libraries make whole new levels of performance possible."

About our manual? "Excellent!" And "...extra fine."

About our BBS for unlimited technical support? "Fantastic!"

And about our competition? Eric Dyke of Inom Software says, "Manx, eat your heart out! Lattice is the right choice!"

Lattice

Professional Programming Tools Since 1981
2500 S. Highland Ave., Lombard, IL 60148

M.A.S.T.
Technical Excellence

"The Matchbox Collection" A POWERHOUSE OF PERIPHERALS.

MASSIVE MASS STORAGE FOR THE AMIGA 500/1000/2000

Maybe you want the unrivalled FLEXIBILITY of TINY TIGER — our UNIQUE standalone SCSI system, compatible with all Amigas.

POWERUSERS may prefer FIREBALL for their Amiga 2000. This true DMA, DRIVE ON A CARD, is designed to burn the competition on **price** and **performance**.

All MAST hard drive systems utilize State Of The Art, 3.5" SCSI drive technology. Both TINY TIGER and FIREBALL include a BUDGET and PERFORMANCE PLUS line of drives.

The BUDGET range comprise Seagate 30 & 50 megabyte drives.

The PERFORMANCE PLUS range feature FUJITSU 3.5" SCSI mechanisms — an indication of the reliability you should expect from this product. Naturally as another quality MAST product, it is SMALL, LOW POWER and covered by our ONE YEAR limited warranty.

PERFORMANCE PLUS drives are available in 45, 90, 135, and 180 megabyte configurations. They are voice coil, fast, and feature their own 24k byte RAM buffer for even faster access. The 45 meg mechanism is SUPERSMALL — at only 1" high it is the same size as our miniscule UNIDRIVE floppy.

TINY TIGER™
THE PURR-FECT SCSI SYSTEM

IF YOU CAN PLUG IN A PRINTER
THEN YOU ALREADY KNOW HOW TO
INSTALL A TINY TIGER
— Simple as that!

★ includes SCSI interface, drive and power supply
—all in one neat little case

★ drive is preformatted with workbench 1.3 installed
—power on and away you go

★ plugs into A500/A1000 and A2000
—great if you upgrade and ideal for moving software from one computer to another

★ connects thru 8 bit printer port
—a natural for SCSI — its 8 bit as well

★ fast — sure is — try this:
copy all of workbench 1.3 from RAM: to RAM: — you can't get much faster. It should take 22.5 secs. To do the same from Tiny Tiger to RAM:=27.6 secs. IN REAL WORLD TESTS — TINY TIGER PERFORMS.
— at lowest price per megabyte

★ printer is passed thru — transparently

★ pass thru for additional SCSI devices

★ TINY TIGER is attached by flexible cable and can be located in a convenient position
— DOES NOT OBSTRUCT USE OF A500 KEYBOARD

★ Bus expansion is left free for other important peripherals
— like memory

FIREBALL™
HARD DRIVE ON A CARD
FOR THE
AMIGA 2000

★ SCSI INTERFACE AND DRIVE
★ TRUE DMA
★ AUTOROOT
★ VERY FAST
★ LOW COST
★ INDUSTRY STANDRAD 50 meg CONNECTOR ON SCSI PASSTHRU

30 meg to 180 meg

CAPACITY	FIREBALL	T/TIGER
30 meg	$ 599	$ 629
50 meg	$ 669	$ 699
45 meg	$ 799	$ 799
90 meg	$1099	$1129
135 meg	$1369	$1399
180 meg	$1569	$1599

REMEMBER — T T PLUGS
INTO ALL AMIGAS FOR A
CLEAN, UPGRADE PATH
— DON'T LIMIT YOURSELF
★ BUY TINY TIGER ★

THE ULTIMATE EXTERNAL FLOPPY DRIVES

UNIDRIVE

★ SUPER SLIM LINE
★ VERY LOW POWER
★ NO CLICKING
★ MAY BE SWITCH DISABLED
(TWIN DRIVE HAS TWO SWITCHES)
★ QUALITY FUJITSU MECHANISM
★ POWERED FROM AMIGA™ OR OPTIONALLY AVAILABLE PLUGPAK

UNIDRIVE $159

TWINDRIVE

★ 1 YEAR LIMITED WARRANTY
★ UPGRADE POLICY—BUY A UNIDRIVE AND HAVE IT FACTORY UPGRADED TO A TWINDRIVE ANYTIME
★ REMOVABLE CABLE
★ ROBUST BEIGE METAL CASING
★ HINGED DUST COVER
★ QUIET OPERATION

TWINDRIVE $299

MEMORY TO GO!
NEED MEMORY FOR YOUR A500/A1000
TRY THIS — MINIMEGS™

2 MEGABYTES OF EXTERNAL RAM
in a very small package

NEW LOW PRICES

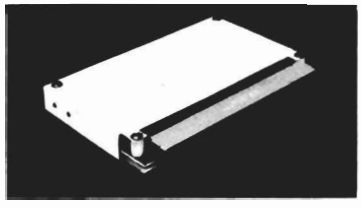
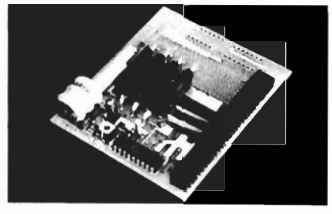


PHOTO OF A500 VERSION

- Auto Configure
- Low Power
- 1 Meg DRAM's
- RAM Tachometer
- Auto Configure LED
- Very Small Size
- Zero Wait States
- Compatible with A501

512K $329 **1 MEG $489**
2 MEG $779 (add $10 for all A1000 versions)

MICROMEGS™ 512 K RAM AND BATTERY BACKED CLOCK



A500 ONLY
$169

This alternative to the A501 is only half the size and uses a fraction of the power — important for power conscious A500 owners.

M.A.S.Terprint™ $1695

300 D.P.I.
DESKTOP PAGE
PRINTER

- FUJITSU ENGINE
- LED TECHNOLOGY
  — for increased
  — reliability
- SMALL FOOTPRINT
  —only 16½"x16½"
- DUAL PAPER FEEDER
- HP-PLUS®

M.A.S.Terprint PS™ $3495

This is the POSTSCRIPT® version of the page printer. It has 35 fonts and 2 meg of RAM (optional 4 meg). Additional POSTSCRIPT font cartridges can be plugged into the printer.

IMPORTANT: MASTerprint can be upgraded to MASTerprint PS anytime — lets you save up for POSTSCRIPT.

AMIGA 2000 OWNERS
OUR NO-CLICK INTERNAL
DRIVE IS A GREAT
DEAL AT $109

MAST PERFORMANCE PLUS HARD DRIVES AND ALL OUR FLOPPY DRIVES FEATURE FUJITSU MECHANISMS

FROM JAPAN'S LEADING COMPUTER COMPANY

Amiga, A500, A501, A1000, A2000, Fujitsu HP+ and Postscript are trademarks of their respective companies. All prices and specifications subject to change without notice.

MEMORY & STORAGE TECHNOLOGY INC.

M.A.S.T. U.S.A.
3881-E Benatar Way
Chico, CA 95928
(916) 342-6278 Tel.
(916) 342-6281 FAX

M.A.S.T. U.K.
Unit 1, Dippen
Brodick, Arran
Scotland, KA278RN
(077082) 234

M.A.S.T. AUSTRALIA
178 Pacific Hwy.
St Leonards NSW 2065
(02) 436-2600 Tel.
(02) 437-4919 FAX

Volume 2, Issue 3

Publisher
Antony Jacobson

Vice-President Operations
Jeannie Lawrence

Editors
Nick Sullivan
Chris Zamara

Assistant Editors
Tim Grantham
Malcolm O'Brien

Contributing Writers
Steve Ahlstrom
Richie Bielak
Ben Blish
Daniel Brookshier
Anthony Bryant
Jim Butterfield
Betty Clay
Don Curtis
Matt Dillon
Andy Finkel
Brad Fowles
Eric Giguère
Cathryn Graham
Eric Graham
Chris Gray
Derek Grime
Jay Gross
Mike Hubbartt
Eugene P. Mortimore
Bryce Nesbitt
Rob Peck
Larry Phillips
Dan Schein
Steve Simpson
John Toebes
Victor A. Wagner
Doug Walker
Clark Williams
James D. Williams
Doug Winger
David Wood

Cover Artist
Louis Markoya

for the AMIGA Transactor
The Magazine for Amiga Programmers

Colour Video Displays 25
by Ben Blish
What do we really need in a computer video display? Ben has a proposal.

Getting A Handle On File Systems 30
by John Toebes and Doug Walker
More technical details concerning the black art of writing a file system.

Programming in M2AMIGA 38
by Anthony Bryant
When you need the power of assembler routines in your M2 code, you need to know the interfacing protocols.

AmigaDOS Patterns In Modula-2 42
by Clark Williams
The origin of AmigaDOS pattern-matching characters, and code to implement pattern-matching in M2

Dynamics For Animators, Part 1 50
by Cathryn and Eric Graham
To make realistic animations, you'll need a bit of mathematics - here are some calculus routines to get you started!

First Steps With C++ 56
by Richie Bielak
What's all this talk about C++ and object-oriented programming? This introduction explains both.

A Sampler From AmigaTEX 64
by Dick Barnes
We are pleased to welcome Dick Barnes, distinguished editor of the late, lamented *Amigan Apprentice and Journeyman*, to these pages. In this issue, Dick provides a graphic demonstration of LATEX, the AmigaTEX enhancement from Radical Eye Software.

Manipulating AmigaDOS File Names 78
by Nick Sullivan
Some convenient C functions for everyday file name handling.

Departments and Columns

Amiga Dispatches 4

by Don Curtis

The ripple effect: Is what you want always just what you need?

ViewPort 8

by Larry Phillips

Has Larry gone off his rocker? Here's the scoop on user interface standards and... Ice cream?

ACCESS - The best of non-commercial software 10

by Steve Ahlstrom

A bevy of great programs, new and old.

Mastering The Raster 14

by Derek Grime

Computer animation can be a powerful presentation tool - but don't get carried away!

Assembly Language Programming 18

by Jim Butterfield

Memory allocation, from the memory pool or the stack.

Hard Copy 68

by Tim Grantham

Two books: *UNIX: The Complete Reference*, and *An Introduction to Object-Oriented Programming and C++*

Reviews

Guru's Guide #1 63

FlickerFixer 70

Lattice C Compiler Version 5.02 72

Cygnus Ed Professional 76

About the cover: *The Fractal Gallery* by Louis Markoya

Louis Markoya is an engineer independently experimenting as a computer artist. His fine art background includes several one-man shows and an apprenticeship with Salvador Dali. He can be reached on PLINK as L.Markoya, CIS as 75766,504 or through *Transactor for the Amiga*. Louis says of this month's cover:

"*The Fractal Gallery* is a *Turbo Silver* raytrace, untouched by any other program except to add the name. However, several other programs were used to create the picture. The walls, floor, ceiling, pictures and spheres have all been mapped with fractals generated with *HAMandel*, a 256-colour HAM fractal generator from MegadeM. In the glass case on the marble stand is a Serpinski gasket, formed with *Analytic Art*, brought into Silver as a stencil, then mapped as a brush and given the characteristics of a lamp, which causes the green-hued glows about it. The Koch snowflakes on the back wall will be recognizable as objects generated using programs from Eric and Cathryn Graham's earlier *TransAmi* articles. The columns and pedestal are wrapped in marble and the slight reflection of the floor and back walls is created by using the 'glossy' feature."

Transactor for the Amiga is published monthly by Croftward Publishing Inc., 10-85 West Wilmot Street, Richmond Hill, Ontario, L4B 1K7. ISSN# 0838-0163. Canadian Second Class Mail Registration No. 7690, Gateway-Mississauga, Ont. USPS Postmasters: send address changes to: Transactor, PO Box 338, Station C, Buffalo, NY, 14209.

Croftward Publishing Inc. is in no way connected with Commodore Business Machines Ltd. or Commodore Incorporated. Commodore and Commodore product names are registered trademarks of Commodore Inc.

Subscriptions: see order card at centre of magazine for complete ordering information and prices.

Send all subscriptions to: Transactor Subscriptions Dept. 10-85 West Wilmot Street, Richmond Hill, Ontario, Canada, L4B 1K7, (416) 764-5273. Note: Subscriptions are handled at this address ONLY. Subscriptions sent to our Buffalo address (above) will be forwarded to our Richmond Hill HQ. For best results, use the postage paid card at the centre of the magazine.

Quantity Orders: In Canada: Ingram Software Ltd., 141 Adesso Drive, Concord, Ontario, L4K 2W7, (416) 738-1700. In the USA: IPD (International Periodical Distributors), 11760-B Sorrento Valley Road, San Diego, California, 92121, (619) 481-5928; ask for Dave Buescher.

Editorial contributions: writer's guides are available by writing to the Richmond Hill address above. Write to the editorial department (same address) with article proposals or articles for consideration.

All material accepted becomes the property of Croftward Publishing Inc., except by special arrangement. All material is copyright by Croftward Publishing Inc. Reproduction in any form without permission is in violation of applicable laws. Write to the Richmond Hill address for a writer's guide.

The opinions expressed in contributed articles are not necessarily those of Croftward Publishing Inc. Although accuracy is a major objective, Croftward Publishing Inc. cannot assume liability for errors in articles or programs. Programs listed in Transactor for the Amiga, and/or appearing on Transactor for the Amiga disks, are copyright by Croftward Publishing Inc. and may not be duplicated or distributed without permission.

Transactor for the Amiga is produced using Amigas and the following products:

Professional Page
Deluxe Paint II
Transcript
Digi-View

Final output by Vellum Print & Graphic Services, Inc., Toronto

Printed in Canada by Bowne of Toronto, Inc.

Amiga Dispatches

The ripple effect and drive select logic

by Don Curtis
Copyright © 1989 Donald A. Curtis

Don Curtis is a police officer in Denver, Colorado, who has for the last three years been assigned to program design and development along with system design and maintenance of ten AT & T Unix 3B2 computers. In his spare time, Don is an assistant sysop on CompuServe's Amiga forums.

Time sure flies when you're having fun... the last few weeks have been very hectic at work. We've been taking several major database applications from the city's IBM mainframe and moving them onto our in-house Unix minis. There is nothing alike about the two environments. The only common ground was the data itself, and even that required extensive manipulation moving it from one system to the other.

All the programs had to be rewritten from scratch, all the users' screens had to be redesigned, and all the concepts had to be rethought. It's been a project that has spanned several years, but most of it all came together the last few weeks. As with all new systems, problems developed.

So what's all this got to do with the Amiga? Well, not much, it's just that when those problems develop, guess whose job it is to fix things? Yup, mine... and that means I've been working my tail off trying to prepare the new machines, move the data over, fix the problems and maintain the system. And that points up some things relating to computers in general, including the Amiga.

When you deal with computers and computer programs, you must assume the end user knows nothing about computers at all. You must also assume that if they can do something wrong, they will. This doesn't mean that end users are dumb or malicious. It just means that they don't necessarily understand every detail of their interactions with the machine they're using. In fact, end users shouldn't *have* to know what's going on... they should only be concerned with the end product. Can the computer and/or program do what the users want it to do, in a timely fashion?

But not only that, do the users understand the consequences of changes they may request in the system? What are the ripple effects of a small change here and there? Let me give you an example:

Here in Denver, our main streets are laid out in a North-South/East-West grid for the most part. In the downtown area, however, the grid is skewed 45 degrees to the rest of the city. Down town streets are laid out in a NorthWest-SouthEast/NorthEast-SouthWest grid. If that's not enough, both the down town streets and many of the East-West streets have the same name, but different suffixes. That is, downtown there's a 16th Street, while out of the downtown area, there's a 16th Avenue. Moreover, if there's no prefix to the street name, the defaults are North or East depending on which way the street runs, unless it's a downtown street and has no prefix! The confusion doesn't stop even there, but there's no need to go into all the gory details. Suffice it to say that the address "510 16th" is useless. It could be any one of three places in the city.

Our users wanted to be able to look up all incidents that may have occurred in a given city block. That's no problem, but one user wanted to make it easy on himself. He wanted to be able to look up, with a single query, all the incidents in the 500 block of 16th, say, whether it was E. 16th Avenue, W. 16th Avenue or just plain 16th Street. That sounds like a good idea - we often get information third or fourth hand that person *x* was seen on *y* street on a certain day, and that it might be useful to see if the police made contact with him. The programmer (an outside consultant) liked the idea also, so he added code that inserted a wildcard character into the place the street suffix would go if the user didn't enter a suffix. That way, if you just said "500 block of 16th" rather than "500 block of W. 16th", you'd get all incidents on any of the three matching streets.

Great! But what no one thought of was the fact that using a wildcard in a query forbids the use of an index (using the SQL query language interface)! If I entered a general query about the 500 block of 16th, the program would scan the entire database (several hundred thousand records) looking for addresses that matched the wildcard. And since downtown streets have no prefixes, you couldn't look up downtown incidents only, you had to look for them all - the wildcard was automatically inserted.

What was the fix? I removed the automatic wildcard and made the system look for exactly what the user entered. Now the system was able to use its index for the search; query times went from 20 *minutes* to 10 *seconds*! You could look for all

three locations by exact matches in less than a minute, far more efficient in the long run. I went back and explained this to the user, and he was perfectly happy. He had seen the long search times and, now that he understood why, he was satisfied.

This was a classic case of the programmer not understanding the user or the consequences of his actions. All the user really wanted was a quick way to look up generic addresses. The key word was 'quick', but the programmer instead thought of 'easy', and wrote the code forgetting, or ignoring, how it would affect other types of searches.

And the ripple effects? It took me hours to find the problem, but a single keystroke in an editor to fix it. The users went from being unhappy about the length of time certain things took, to being happy with the change.

Remember this little story the next time you wonder why things don't always seem to work the way you'd like. Also remember it the next time you feel like flaming Commodore for not producing the exact feature you want in an Amiga.

If you write a program for yourself, you can customize it to your heart's content. You are only concerned with your usage of the program for your purposes. You don't care if someone else likes your method since no one else will use it. Add a second potential user, however, and you have a possible conflict between your way of doing things and their way. The answer is a compromise. Obviously, the more potential users and/or uses of a computer or computer program, the greater the potential for conflict. You can give the users choices to help let them customize the program for their own purposes, but you can't make it perfect for everyone. That's just not possible.

The same is true for hardware, to a lesser extent. The base hardware of the machine pretty much has to be the same for all users. Compromises have to be made, and some of them may not meet with your approval. Take the standard high resolution screen of the Amiga. It's in interlaced mode to make it conform to TV standards. That way, you can take the output and put it on the air, tape it or whatever, with minimal processing. That's good... for folks interested in video applications. It's not necessarily good for the rest of the Amiga community. As you know by now, using interlaced mode with a short persistence monitor (as the vast majority of monitors are) causes screen flicker. Screen flicker can be annoying... well, no, it *is* annoying.

So why did Commodore decide to compromise in favour of interlaced screens rather than non-interlaced screens? Surely the majority of users aren't video producers are they? I can't give you the 'official' reasons for deciding on interlaced mode. But suppose Commodore had made the opposite choice...

When the Amiga first appeared, its graphics capabilities rivalled those of any computer available for under $50,000 or so. Video Tape Recorders had become common home appliances. Desktop video was almost unheard of, but the potential was there. With the Amiga, a home user could suddenly produce 'Saturday Morning' quality cartoons at home, *but* for the home user to take advantage of this, the video output had to be able to be recorded - it needed interlace mode. The alternative was to require an expensive video adapter, as with other computers.

With interlace, it was all there. With its custom graphics chips, large number of colours, 640 x 400 resolution and the ability to directly tape the output, the Amiga was a natural for video applications. So much so that not only did the home users pick up on it, but so did many professional video shops. If you think that wasn't important, go out and try to find an Atari ST - in some ways a similar machine - running a video application. If you find one, let me know where. I'd be interested in how they did it, what they had to add to the machine to make it work, and why they didn't choose the Amiga in the first place.

Without interlace, the Amiga was still a wonderful computer, but it wouldn't have fit all the niches it does now. I don't believe that it would have sold as well as it has, and apparently neither did its designers.

That's the kind of consequences I'm speaking of. The initial buyers of the Amiga were hackers and folks interested in its video capabilities. It took a while for others to catch on and see the other features that made the Amiga stand apart from anything else on the market. If the initial buyers hadn't been there, the Amiga may not have made it. Commodore wasn't in very good financial shape back then and the Amiga's success has been a major reason that they have been able to recover.

Lots of decisions like that went into the design of the Amiga - the choice of interlaced versus non-interlaced video modes was only one. Even seemingly trivial things like the colour of the case had to be decided from a range of available choices. When you get upset with Commodore for something you don't like, try to think of the consequences of having it some other way. When you put it all together, I think you'll find that they made some pretty smart decisions after all (along with the dumber ones) when the total effect on the user community is considered.

Disk drive selects

That brings up another whole discussion. Why did Commodore make the external drive on the A2000 drive DF2:, and why did programmers hard code their programs to look for DF1:?

A little background is needed here on how drives are selected (accessed) on the Amiga. In simple terms, on any disk drive, floppy or rigid, there's a control line called 'Drive Select' that's used to make the circuitry active. Most computers have the ability to control several disk drives, and the Amiga is no exception. They do this by applying a signal to one of many select lines, thus telling the disk drive attached to that line, "Get busy!". The general methodology is to use a jumper on the drive itself, or to have a twist in the drive control cable so the select lines attach differently. The result is that only one select line is actually hooked to the drive's circuitry. The rest are essentially unconnected. Thus the drive only becomes active when that

particular select line is chosen. If you didn't do this, what you'd have is that when you attempted to access one drive, all the drives would turn on simultaneously and nothing would get done... the system would become hopelessly confused.

To add a drive to a circuit like this requires the user to know how to set the select jumper, or twist the cable. While moving a jumper is fairly simple (usually you just pull a jumper off of one set of pins, and place it on an adjacent set of pins), many users are afraid to open up the drive and 'play' with them... so they have a service shop do the installation for them. That's added expense, time and trouble.

The Amiga used a different philosophy. It was decided that they would have their circuitry set up so that you could take any drive, and put it at any position in the drive chain, and not have to make a jumper change. All the drives are jumpered identically as "Select 0", the first drive in the circuit. Now the Amiga has the same multiple select lines coming out of the controller chip (8520), labelled Select 0 through Select 3. The first line (Select 0) is always attached to the internal drive... thus it's drive DF0:, but Commodore performs a little magic on the other three select lines to allow the drives to all be the same.

If you'll look in your Amiga manual, you should find a diagram of the disk drive connector. You'll see three pins labelled *Sel_1* through *Sel_3*. This is true for the connectors on the drives themselves also. On the Amiga A1000 and A500, the external connector has all three lines hooked up. On the external disk drives, all three lines are hooked up on the input connector, but only two of the lines are hooked to the output connector!

On the drive, the line attached to the *Sel_1* input connector is attached to the drive itself as the "Get busy!" line. The remaining two input lines are passed to the output connector, but shifted one position. That is, input line *Sel_2* gets connected to output connect line *Sel_1*. *Sel_3* goes to *Sel_2* output, and there is no connector on the output line *Sel_3*. See the magic? If you have three drives on your system, drive DF2: should be activated by the computer's Select 2 line. When the computer activates Select 2, the signal gets passed out the back port into the first drive's *Sel_2* input. From there, it simply gets hooked to the *Sel_1* output and passed on to the second external drive. The second external drive gets it on the *Sel_1* input, so it's connected to the drive itself and activates the drive.

That's similar to twisting the wires, but the system does it all for you... you don't have to worry about it. However, the A2000 is a slightly different animal. It can have a mix of drives inside it. There are several combinations possible, and one of those combinations is two internal floppies, DF0: and DF1:.

Since that's possible, they couldn't run the Select 1 (select line for DF1:) line outside the machine - that would have made it possible to have two drives attached to the same select line, and nothing would get done because both drives would become active at once.

Because of the design of the external drives, it wouldn't work to have you move the jumpers around. The lines wouldn't be passed properly. So a decision had to be made. Either design the A2000 to use a completely different drive set-up, or have the first external drive recognized as DF2:.

The consequences of the first choice would have meant that folks who moved up to the A2000 from the A1000 or A500 wouldn't be able to use any existing external drives they had. Dealer's would have had to stock two different types of external drive, and the users would have had to learn how to set drive select jumpers or pay a shop to do the job for them.

The second choice made much more sense. There's nothing magic about a drive addressed as DF1: - the only drive that's intrinsically special is DF0:, which you need to boot the machine (prior to autoboot that is). The users could keep their current external drives and just plug them in. The drive chaining logic would continue to work, and the 3.5" drive bay could still be used for either a Hard Drive or a second 3.5" floppy.

So what was the problem? Some software authors have hard-coded their programs to look for a DF1: drive specifically! Those programs never look beyond DF1: to see if there might be a DF2:. Prior to the A2000, that was a safe assumption, but now the second drive can be either DF1: or DF2:. And whose fault was that? Was Commodore wrong for allowing a second drive but no DF1:, or were the software writers wrong for making assumptions about how the machine is configured?

Commodore has stated from the beginning not to make assumptions about the configuration of the machine beyond 256K and a single floppy labelled DF0:. At this point it may seem safe to make assume 512K, but even that's not true. There are still some 256K A1000s out there. That leaves the fault squarely at the feet of those software authors who made assumptions about the machine's configuration. And that comes right back to the idea of changes and ripple effects.

A simple change in design broke some programs because the authors chose not to follow the rules. How hard is it to let the users define their system to the program, rather than have the program define the system to the users? It's the responsibility of software authors to allow the users the widest possible freedom of choice. It's also the responsibility of the authors to have their programs go out and look to see what resources are available. Imagine how absurd it is to have $4,000 of computer sitting on your desk, yet a simple program won't run because you don't happen to have a drive with the name "DF1:" in the device list. Even though there work-arounds do happen to exist for this particular problem, the point is that there should *be* no problem - and that there is a general lesson for software authors to learn from this specific example.

Well... I guess it's time for me to get off my soapbox. As I write this, the Spring COMDEX is rapidly approaching. Nothing spectacular is expected (well... maybe one thing) but I'll update you on that next issue. Until then... enjoy!

HiSoft BASIC Professional...The Power to Create

HiSoft BASIC Professional is a fully-integrated compiler and editor that makes it easy to convert all your AmigaBASIC programs into faster-running machine code; in fact, it is so simple and quick to use that you will soon find that you no longer need the AmigaBASIC interpreter, you can do everything you want with **HiSoft BASIC Professional**. Amiga libraries can also be accessed as if they were built-in statements allowing complete machine access.

HiSoft BASIC Professional includes full MENU support, with event trapping and powerful sprite routines, using the OBJECT keywords.

Compiled programs can execute in their own window(s) or use the CLI window for minimum size.

HiSoft BASIC Professional is a no-limits language; string variables may be up to 16Mbytes in length and there are no limits on array sizes either (subject to available memory). Code generated is fully 68010/020/030 compatible.

Compiled programs have no run-time overhead. Compiled programs multi-task, of course, but as they all share the same library, memory usage and disk space is kept to a minimum. There is also an option to produce completely stand-alone programs which include only as much of the library as they need.

Features include:

✔ Compatible with Microsoft QuickBASIC 3 on the PC, Atari ST BASIC, AmigaBASIC and most other BASICs,

✔ Full structured programming.

✔ Optional line numbers, many compiler and run-time options allowing tremendous flexibility both at compile & execution time,

✔ Full recursion in sub-programs and functions plus local and global variables, arrays as parameters,

	HiSoft Professional™	AC BASIC™
SIEVE		
Compile time (secs)	7	20
File size (bytes)	868	2228
Run time (secs)	1.0	2.7
CALC		
Run time	1.6	4.9
BSPREAD		
Compile time	55	80
File Size	35816	56036

HiSoft BASIC Professional *- now BASIC programming on the Amiga has come of age.*

Order your Copy Today at $159.95, and take a look at Devpac Amiga, the best Assembler available for the Amiga. Together they make an unbeatable programming team!

MichTron®
576 S. Telegraph, Pontiac, MI 48053
Orders and Information (313) 334-5700

NOW WE HAVE IT ALL

AMIGA....C128....C64....IBM....APPLE

Software/Hardware.....New/Used

We now carry a varied line of SOFTWARE and HARDWARE for virtually every Major Home Computer SYSTEM!

JUST LOOK AT THESE TYPICAL EXAMPLES:

USED IBM HARDRIVES & MONITORS From $99

PLUS : (Prices Below INCLUDE SHIP & HANDLING)

PUBLIC DOMAIN Disks ..... $ 5.00

CHEAT-SHEET (SOLUTIONS TO MANY TOP GAMES!) ..... $ 5.00

SPECIAL: DRAGON'S LAIR ON VHS TAPE!
ENTIRE GAME LEVELS ON TAPE. SEE THE MOVES TO WIN. (INCLUDES PRINTED Step-By-Step GUIDE!) $12.95

ALSO NEW & USED Cables, Controllers, Expansion Cards, Disks

PLUS - Special Services.....DIGITIZING.....DOCUMENTS-ON-DISK

COMPUTER HELP & TIPS . . . MORE! CALL or WRITE to:

CAPPCO

2943 Ridgeway Dr.
National City, CA.92050
(619) 470 - 4580

GAME PROGRAMS WANTED NOW !

Attention Amiga Developers and Programmers.

Virtual Reality Laboratories, Inc. is looking for a few excellent new entertainment and educational games for the Amiga. To be accepted, these games must excell in graphics and sound capability. If you have some work that fits this description, send for guidelines to: VRLI, P.O. Box 609, Atwood, CA 92601.

Please don't submit until you have read the guidelines.

ViewPort

*Standards, yes... but **flexible** standards*

by Larry Phillips
Copyright © 1989 Larry Phillips

Larry Phillips is an Amiga hardware and software hacker from Vancouver, British Columbia, and a sysop on CompuServe's Amiga forums.

"I'd like some strawberry ice cream please."
"Sorry, we only have vanilla."
"No choice?"
"No, but vanilla is the best flavour. Absolutely the only flavour you need. It is cold, a nice yellowish white colour, and won't offend the taste buds like some flavours. And best of all, it makes choosing easy."
"But what if I don't like vanilla?"
"But everyone likes vanilla. That's why we serve it. If you really don't like it, you could make your own with our handy ice cream maker."

Ice cream in a computer magazine? That's it, you say, Phillips has really cracked up now!

But does the conversation ring a bell? I was going to make the ice cream apple-flavoured, but that would have been too obvious. The unlikely scenario above is how I perceive the mentality of those who would impose their world upon us under the guise of a 'standard user interface'.

I wouldn't mind so much if the proponents of a standard user interface were to stop short of the details of the implementation. They don't though. They start talking about whether or not a file requester should sort while loading, sort on demand, or not sort at all. They would dictate to us what the gadgetry should look like, whether double-clicking on a file name should be the same as clicking on the OK or LOAD gadget, whether you should select the parent directory with a gadget or as part of the file list. They would dictate the smallest detail, all in the name of a standard user interface.

It is frustrating at times to talk to those who hold the Mac interface near and dear to their hearts. Is isn't the fact that they like a standard user interface that bothers me, because I agree totally that a standard is desirable, but the fact that they come from a background where it is considered acceptable for one person, or group of persons, to dictate what that standard should be, right down to the most niggling little details.Those programmers who feel the standard isn't quite what they want in their program are, of course, free to write their own user interfaces. Whether the program that results from this will sell or not is another thing altogether. There is a fair amount of pressure from Apple, from Mac users and from other Mac developers against any changes to the user interface. Still, the programmer is certainly within his rights to make his program behave in any way he sees fit.

There are many problems inherent in this situation and, though it does have its good points, I feel that there is no reason we cannot take the good points without the problems, and apply them to the Amiga. A common wish among Amiga owners is for a file requester that is built into the system, so I will use this as an example of what might be done to promote a standard, while eliminating the problems.

The primary benefit of a standard user interface is that the new user can easily pick up any application and immediately know that it operates, for the most part, just like all his other applications. This reduces FUD (Fear, Uncertainty, and Doubt), and allows both novice and veteran alike to quickly become familiar with a new program.

The problems really boil down to one thing. If you, as a user of the machine, don't happen to like the user interface, or any part of it, that's too bad. You are stuck with what you get. If a programmer decides that he wants to write his own file requester, and you buy the program, you are stuck with that, too. The only difference being that in the case of the system-supplied file requester, you are stuck with the official version, and in the second case, you are stuck with the programmer's version.

The first questions we might ask are:

"Who is the standard for?"
"Can we figure out a way to have a standard and, at the same time, have choices?"

The answer to the first question, in my opinion, is that the standard is for the user. That it makes things easier for the programmer to have a built-in file requester is also a good thing but, in the final analysis, the benefits are to the user.

The answer to the second question is a resounding 'Yes!', and it is this that I want to address in the following proposal. I am making the assumption that CBM will provide, in some future release, a built-in file requester; here I want to discuss a possible implementation of it. This proposal does not concern itself with the details of how the user interacts with the file requester. It does not deal with the look, feel, number of gadgets, or size. It deals only with the interface between the calling program and the file requester itself, and you will see why in a moment.

The general idea is that a built-in file requester should be configurable in some ways, through the use of user-settable flags. It should be callable by any program, easily, like any other system routine. It should provide all the information expected from a file requester, which is to say that it should provide a path and a file name (or multiple file names) to the calling program.

The file requester should also be *replaceable*, easily, by the end user. Consider the implications of replaceability. A programmer, given an easily callable file requester with adequate functionality, would be hard pressed to come up with a reason to roll his own and to include it in his program. The only reason programmers of Mac and Presentation Manager applications even consider doing it is because they feel that the file requester provided by the system is inadequate, ugly, or both.

Any inadequacy can be overcome by providing a calling convention that is both complete and extensible. Any ugliness can be overcome by allowing easy replaceability. Completeness of the calling convention is the most difficult part of this scheme when you consider that the convention must allow for all 'proper' parameters or flags likely to be needed, for both now and in the future. Programmers, being what they are, would undoubtedly come up with a flag or parameter they simply must have, just as the ROMs are in full production and ready to ship. The second part, the facility that allows easy replacement of elements of the system, is already in place, with fonts, keymaps, printer drivers, device drivers, libraries, and even individual routines within libraries.

Let's start with the programmer's needs. An application might need to be able to specify that it can accept multiple file names, or that it can only handle one. It must be able to provide title text for the requester and a path name for the directory listing. It must be able to set flags specifying whether it wants the requester listing to filter the directory to show only files matching a pattern, and probably a few I can't think of at this moment. The requester must in turn inform the application if a selection was made, if it was a multiple selection, if the requester was cancelled without a selection, and of course, the path and file names themselves. The programmer need not concern himself with what the requester looked like, whether it sorted a directory, or whether mouse or keyboard or voice input was used to make the selection. Those decisions are best left to the user.

Configurable options, perhaps set from Preferences or a similar program, might include such things as 'hot' selection ability, double-click or 'OK' gadget only, whether the requester comes up in a fixed position or in relation to the mouse, and so on. Options settable in this way would be those that are of no particular concern to the programmer, and are more a matter of taste for the user. The setting would ideally affect any file requester.

This brings up the heart of the matter. The programmer has his system call, and can depend on the results of that call. If the file requester can be replaced easily, the user can have his choice. In this way, we can have a standard in more ways than one.

The first standard is that the system file requester is what the new Amiga owner will see. It will be called for most programs (the exceptions being programs that pre-date the facility). It will provide a common ground for talking about and learning the operation of the system as well as individual applications. The other standard is obtained when a user decides to replace the system-supplied file requester with one that he prefers. The replacement, while it can, in one way, be looked at as non-standard, really is a standard after all, if it is called whenever the system requester would have been called. Thus, all applications calling in the normal manner will call the same file requester without knowing or caring that a replacement is being called, making all applications using it act identically from the user's viewpoint, and in a way that the user prefers it to act. It seems to me that all the advantages of fixed system file requesters are obtained, while still allowing for choice on the part of the user.

Choices are important, since every Amiga owner seems to have a preference for one file requester over another. Personally, I like one that can be dragged around the screen and is small enough to let me see text that might otherwise be hidden. Others prefer one large enough to display a lot of file names. Some like the ability to use the keyboard, or the mouse, or both. Do you like gadgets for selecting a volume name, or do you prefer a textual listing? Do you want assigned directories to show up along with the volume names?

These preferences need not concern the programmer, who can rest assured that, when he calls for a file name, the user will be supplying one in the way that is most comfortable for the user, rather than in a way left up to Commodore or the application.

We need not limit the replacement to any particular appearance; indeed, we need not have it appear on the screen at all. A person with some sight impairment might want a large file requester with large characters, preferring to see the names easily even if only a few are visible at a time. A completely blind user might want a 'talking file requester' that has no visual representation at all. In fact, almost any conceivable method of presenting a list of files could be handled by a replacement.

Remember, too, that although I have talking about a file requester, the same principle could be applied to other user interface building blocks as well. I think we can have standards and still allow creativity. We can have an easy to learn machine that can be changed as the user advances. We can have a machine that is unparalleled in its accommodation to widely varying preferences or needs.

ACCESS

The world of non-commercial software

by Steve Ahlstrom
Copyright © 1989 Steven Ahlstrom

As the primary sysop on CompuServe's Amiga forums, and a veteran Amiga programmer, Steve Ahlstrom sees just about every notable public domain and shareware program that gets written, and in this column he tells you about the ones that have recently caught his eye. If you have some information you'd like to share through this column, you can leave a message to Steve on CompuServe (his PPN is 76703,2006), or send the info to us and we'll pass it on.

Key Map Editor by Tim Friest

One of the Amiga's strongest points is that it really *is* the computer for the rest of us (whatever the Macintosh may claim). The Amiga allows easy configuration in just about any area. By selecting your printer from Preferences, you have configured your computer to work with practically all off-the-shelf commercial products. You may change colours, screen resolution, font sizes, mouse and key speeds, and make these the defaults. Unlike other machines whose ambience implies "Do it our way or don't do it at all", the Amiga puts you in control of your computing environment. Now, thanks to Tim Friest's Key Map Editor, with its ability to create or modify keymaps, there is an additional tool to customize your environment.

What are keymaps? They are the files hiding in the DEVS:Keymaps directory, with names like *usa1*, *gb*, *is* and *dk*. The Amiga is an international machine. According to CBM, 70 per cent of the Amiga's worldwide sales are outside of North America. Keymaps allow the machine easy penetration into worldwide markets. With keymaps, machines do not have to be custom manufactured for the local language/alphabet differences in each country in which they're sold. All that is necessary is that a keymap for that country be written and then loaded from the startup-sequence. For instance, in the *usa1* keymap, there is no British monetary symbol. Load the British keymap (**setmap gb**) and shift-3 becomes the British pound symbol, replacing the # in the *usa1* keymap.

Keymaps thus simplify the internationalization of the Amiga. That is not their only use, however.

When the typewriter was invented, some hundred odd years ago, various keyboard layouts were tried. After much trial and error, disagreement, and confusion, the 'qwerty' system that most of us use became the standard. It is not the most efficient for typists. Different layouts can dramatically improve speed. Why did it become the standard? In those days, the limiting factor was the mechanics of the typewriter. The bottleneck was the physical mechanism that produced a character on a piece of paper from a pressed key: type too fast and the typewriter could easily jam. 'Qwerty' was devised and became the standard precisely because it was inefficient and slowed typists down.

Now, of course, there is no such limitation. Various alternative layouts have been devised that are much more efficient for the typist. The most popular of these is the Dvorak keymap. Once learned, its use will greatly improve your typing speed. Just try to find a Dvorak typewriter, though. Or try to find any other computer than will allow you to change its keymap to Dvorak and have it work with practically all available programs. With the Amiga, it's a cinch: **setmap usa2**.

Hopefully you will agree what great and marvellous things keymaps are, and that the ability to change keymaps makes the Amiga stronger and more well rounded than other machines in its class. You might ask, though, why should I be excited about a Key Map Editor? Since the computer comes with a variety of keymaps and CBM is surely going to create keymaps for any new country the machine is marketed in, what use is such an editor to me? The reason is that pressing one key does not necessarily mean that only one character will appear on the screen. With the Key Map Editor, you can employ the keys you seldom use (or combinations of SHIFT, ALT and CTRL with those keys) for sending practically any textual information you want to your application. Key Map Editor is a very powerful keyboard macro creator.

As a very limited example, I have redefined my numeric keypad. It now 'looks' like this:

(	)	/	*
run dman\r	8	9	-
acc\r	whap\r	6	+
dir\r	2	3	(enter)
uucp:\r uucico -x1 -stcr		.	

AMIKIT IS HERE!

AMIKIT IS HERE!

No more frustrated hours spent staring at your computer screen! No more blundering through the Workbench or CLI -- the ultimate aid for novices and experts alike has arrived!

Step by step, AmiKit helps you explore and benefit from the awesome power of your Amiga. From the most basic to complex commands, AmiKit teaches you how to gain control and confidence using your machine, whether its an A500, 1000 or 2000. And what's more, AmiKit includes over thirty Public Domain software programs, including: Workbench 1.3; a virus checker; a database; a spreadsheet; a word processor; games; and much more!

AmiKit is one of the most effective ways sucessfully to navigate your way throughout the workbench and CLI. As a special introductory offer from Transactor for the Amiga, you can get AmiKit for $34.95* (plus $2.50 for shipping) -- $5.00 off the recommended retail price -- an amazing bargain! When Amiga User International, the leading European magazine, reviewed AmiKit, they were extremely pleased with the package. We know you will be, too!

To: Transactor For the Amiga
10-85 West Wilmot St.
Richmond Hill, Ontario, CANADA
L4B 1K7
I would like to purchase AmiKit at the special price of $34.95 + $2.50 shipping ($37.45)*. Please make cheques payable to Croftward Publishing, Inc., or debit my Mastercard/Visa account:

☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐

Expiry Date _____

Signature _____

Name _____

Address _____

_____

_____

Telephone _____

*ONTARIO RESIDENTS ADD 8%

T9046 *Please allow 28 days for delivery*

Photocopies of this coupon accepted

So far, I've only redefined five keys, and those only in their non-qualified states (that is, without SHIFT, etc).

Instead of displaying a **7** when I press the **7** key on the numeric keypad, my keymap sends **run dman<cr>** to the active window (DMan is a directory utility that I use). Eight keystrokes are saved in running that program. You can get a bit more complex. When the **5** key on the keypad is pressed, my keymap sends **whap<cr>**. **Whap** is an ARexx script (I use William Hawes' WShell, so no preceding **rx** is necessary). **Whap** could just as easily be an AmigaDOS Execute script. Just one keystroke causes an entire chain of events to occur.

Key Map Editor is very handy! Should you wish to create keymaps for various applications, you could have a key on your default keymap that would **setmap** another keymap then run your application. There are a lot of possibilities here. Because of the combinations of the various 'qualifying' keys, each key on the keyboard can respond in up to eight different ways, giving you hundreds of combinations for user-defined macros. Any correctly written program that uses the default keymap (rather than its own internal one) will fully support your modified keymaps.

Key Map Editor is complex, yet easy to use. Everything you need to know is in its documentation.

Key Map Editor is a freely distributable, copyrighted program by Tim Friest. It is available in CompuServe's AmigaTech Library 13 as **keyed.zoo.**

BindNames V1.0 by Dave Haynie

If you have a hard disk, do you find it inconvenient to be constantly editing your startup-sequence file for the logical assignments most new programs require? You add the assignment, reboot, then find out that your assignment is dependent upon another assignment that has not been made yet?

Dave Haynie comes to the rescue with BindNames, a program designed to do all of your logical name assignments at once. You create a sub-directory in your root directory called Names. In this directory you create a file which contains your assignments. For your system assignments, you might have a file in *sys:names* called *system* that looks something like:

```
BIN:     SYS:bin
OS:      SYS:os
C:       BIN:c
COM:     BIN:Com
L:       OS:L
FONTS:   PATH:FD:Amiga,FD:PD,FD:PS,FD:CityDesk
FD:      OS:Fonts
S:       OS:s
DEVS:    OS:Devs
LIBS:    OS:Libs
ENV:     RAM:Env
T:       RAM:T
```

In your startup-sequence you would replace all the assigns to a call to BindNames: **bindnames system**.

BindNames will read all name files before making any assignments. It will figure out dependencies. It does not matter how you order the names. It will create directories it cannot find, such as the RAM:Env and RAM:T in the above example, and will generate warnings for assignments it cannot resolve.

Instead of 'assigning the world' when you power on the computer, you can make some specific assignments only when you need them. If you want to run a program that requires logical assignments, just create a file to put in the **sys:names** directory listing the needed assigns. Before you run that program you would type **bindnames <filename>**. You could go further and create an ARexx or AmigaDOS execute script that will do it all for you - then add the command to your custom keymap!

BindNames, including source code, has been released into the public domain by Dave Haynie. It is available in CompuServe's AmigaTech Library 14 as **bindna.arc**.

KeyHack by Jez San

In Volume 2, Issue 1, I talked about a program called SetCPU 1.4, also by BindNames author Dave Haynie. SetCPU allows you to copy the system ROM into 32-bit RAM (if you are using Commodore's A2620 board). This greatly increases the speed of your system.

Unfortunately, the early keyboards that were shipped with the A2000 are not compatible with SetCPU. These keyboards are relatively easy to spot. They are made by Cherry and have smaller function keys than the later keyboards. If you own one of these keyboards, and an A2620, Jez San's KeyHack program will let you use SetCPU all the same.

KeyHack is a quick, temporary hack, to patch a RAM copy of the Amiga OS to slow down the keyboard handshake.

KeyHack is a freely distributable, copyrighted program by Jez San. It is available in CompuServe's AmigaTech Library 14 as **keyhk.arc**.

MemWatch by John Toebes

MemWatch has been around for a while, but many times we tend to overlook the older programs for the flashy newer ones. Newer does not always mean better. MemWatch is a classic that has an important use for your system.

MemWatch is a relatively simple program that can help prevent random gurus. Run it from your startup-sequence; it will sit in your system quietly, watching for errant programs that change the values in the first 100 bytes of memory. Few programs write to low memory intentionally - it is usually a side effect of inadvertently referencing a null pointer. In many cases, writes to low memory are not harmful. Sometimes,

though, they are time bombs just waiting to welcome the turbaned one. It all depends on what addresses are overwritten.

When it is invoked, MemWatch saves a copy of the first 100 bytes of memory. Afterwards, it periodically compares the current contents of low memory with its stored copy. If any change has taken place, MemWatch restores the correct value then puts up an alert to let you know what has happened. The alert tells you both the new value and the address to which it was written.

To do its job effectively, MemWatch must check low memory frequently, so that any damage can be repaired before it propagates. By default, the program checks low memory every 5000 microseconds. This period is user configurable.

There is no guarantee that MemWatch will catch and repair all damage to low memory, but most of the time it will. It is a very useful program to have running in the background when you are developing your own code - it is an extra debugging tool. It also is useful in finding those incorrectly-written programs that do damage to low memory. Some of them, like Aterm 7.3, write to low memory but do no real damage (Aterm 7.3 writes an 0x04 to location $00). Other programs may do more damage and eventually lead to unexplained system crashes. MemWatch will let you know which programs you might have to watch more closely.

MemWatch is a freely distributable, copyrighted program, written by John Toebes and copyrighted by The Software Distillery. It is available, including both executable and C source, is CompuServe's AmigaTech Library 14 as **mem.arc**.

Guru by Mike Haas, and Guru by George Musser, Jr.

"Guru Meditation" is a very familiar phrase for all of us. For most of us it means just one thing: the system has crashed. We are given cryptic numbers to explain the cause of the crash but, in and of themselves, the numbers are mostly meaningless. There are few readily available sources that contain a list of the error numbers. One is the **exec/alerts** header file from your C compiler, assembler, or from the Native Developers Disk from CBM. Another listing is contained in the GOMF manual. Many people do not have access to either of these lists.

There are a couple of programs that allow you to type in the error portion of the guru number (the number to the left of the period is the error, the number to the right is the address at which the error occurred), and will give you an explanation of what the error means. The explanation may not be all that meaningful, but much more so than just a series of numbers.

Both of these programs are named 'Guru'; they perform similar functions. In most cases, the program by Mr. Haas has better formatted and more informative output; C source for Mr. Musser's program is available. Both programs are freely distributable and can be found archived together in CompuServe's AmigaTech Library 14 as **guru.zoo**.

The fastest
Modula-2
Software Development System for
AMIGA
$ 249
Demonstration disk $ 10

Modula-2 is the successor of Pascal. Both languages were invented by Professor Niklaus Wirth. Modula-2 is an all purpose language suited for educational, home and professional use. Modula-2 leads to good programming technique, easy readability and maintainable programs. In fact, a lot of schools and universities have chosen Modula-2 for their programming courses.
M2Amiga is a software development system for Modula-2 based on an extremly fast single pass compiler. It is fully integrated into Amiga Workbench and is easy to use from CLI or Workbench. It runs on all Amiga models with a minimum configuration of 512k RAM and one disk drive.
M2Amiga has been developed specially for the Amiga and therefore optimally supports the features of this unique computer.
- Single pass compiler produces optimized native MC68000 code from standard ASCII files. Full support of latest edition of Modula-2.
- Fastest library calls possible on the Amiga. No intermediate code needed for library calls and parameter passing.
- Supports FFP, 32 bit IEEE and 64 bit IEEE reals formats, you can even mix all three formats within the same program.
- Easy access to MC68000 registers, allows the insertion of inline code.
- Full interface definition to the Amiga's operating system following the official definitions.
- The best runtime system currently available on the Amiga. Opens and closes all used libraries, captures all GURUs and releases all used resources after program crash or termination. Routines for System requesters enable user interface consistent with the Amiga's Intuition.
- The emacs based editor places the cursor on compilation errors and explains them in full text.
- In just a few seconds the linker produces stand-alone applications that can be run from CLI or Workbench. Stand-alone applications do not contain any useless overhead – they are compact and fast.
- All programs can produce icons for their output files to allow effective use of the intuitive Workbench environment. Experience the new feeling of program development by selecting and double clicking icons!
- Also included are a project management tool and other utilities.
- Besides the handy manual, an introductory guide to Modula-2 is included. This documentation allows both the beginner and the advanced user to find any information needed quickly and easily.
- Demo programs show the use of various features of the Amiga.
- For professional program development special utilities are available.
  + Source Level Debugger, the new way of debugging your programs. Inspect all your variables (displayed in their own format), zoom into structures as records or arrays or follow pointer chains by clicking the mouse buttons
  + Object file converter, converts standard Amiga object files into M2Amiga format. This allows linking of routines written in other languages.
  + Linker for devices or libraries. Add power to the Amiga by supplying new libraries and devices. Easy and intuitive programming instead of fooling around with Assembler and Amiga linker are possible now!
  + Modula-2 Amiga Programming System Environment

We have Modula-2 Compilers for HP-UX, IBM/370, PCs (Taylor, M2SDS, JPI), OS-9 and Sun, tomorrow even more!
Texas residents add 8% sales tax. International Orders add $ 20 shipping.

The Modula-2 people:
INTERFACE TECHNOLOGIES
3336 Richmond, Suite 323, Houston, TX 77098-9990 (713) 523 8422
Dealer inquiries welcome 1-800-922-9049

2U
A. + L. Meier-Vogt
Im Späten 23
CH-8906 Bonstetten/ZH
Switzerland
Tel. (41) (1) 700 30 37

Mastering The Raster

Motion and emotion

by Derek Grime

Derek Grime is president of Beyond Graphics, a Toronto-based studio that uses the Amiga for a variety of graphics, animation, and video applications. In this regular column, Derek talks about using the Amiga for professional work. Derek can be contacted at Beyond Graphics, P.O. Box 1249, Station F, Toronto, Ontario, Canada, M4Y 2V8.

Last fall the respected Canadian design magazine, *Studio*, held its annual awards honouring excellence in the graphic arts. It was the first year that computer graphics were added to the large list of categories. The response was tremendous. Each computer entry was studied as the judges looked for the one piece that stood above the rest. Finally the choice was made; it was unanimous.

Nothing deserved an award.

The judges felt that the computer-created work was so inferior to the other categories, they worried that it would bring down the high standards set by the other fields. Unfortunately, this was not a fluke. The judges were well known designers, and adding computer graphics to the awards was their choice. They just found that such an exciting technology was being misused. This attitude is widely held by professionals.

Last month we discussed aspects of bad design as it related to colour and type. This month we'll apply some of the same techniques to computer animation and video on the Amiga. Moving graphics around adds an entirely new set of design problems to an already difficult job.

As a computer assisted video maker or animator, your mandate is to get the client's idea across in the best possible way. As a small business person you have to get the work finished quickly. This means that you may not be spending as much time on the creative extras as you would like. Believe it or not, your videos will often look much better for it.

Less is more

As in all forms of traditional design the 'less is more' statement is particularly important when applied to computer graphics. The customer comes to you because you can present his concept in the most straightforward manner. We know that computers are capable of incredible effects, but your clients often don't. Rarely does a job come in where you are allowed to go wild.

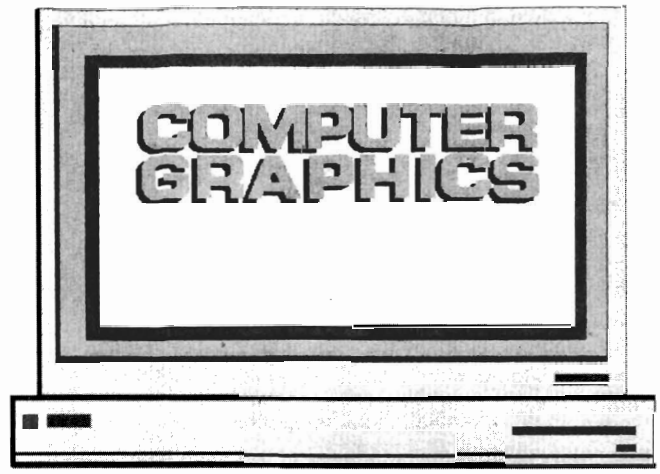
A typically simple industrial video job would contain several sales charts that would grow and then fade from image to image. You'll find that Deluxe Paint and TV Show will get a lot of use. These two programs (or other flavours in the same family) do about eighty percent of the production work in the average Amiga studio.

Sometimes the client may want a look that you would have trouble reproducing on the Amiga. As always, honesty is the best policy. The customer should be made aware of the limits of your studio. You don't have to tell them outright what the specs are; believe me, they're not interested in the technology. It doesn't matter to them what processor you use or how many bit planes you've got. They have no interest in the fact that you are using an Amiga over a Mac II or an IBM. We know we can't give them a smooth shaded background of two hundred shades of blue. We can't easily produce text that will still look great when blown up to room-size from a slide. I'm always discouraged when someone else promises a job that is well beyond the capabilities of the system. It turns clients off using the desktop video producers and damages the Amiga's reputation.

Don't fall into the trap of being too eager to please. Show them your portfolio video. If it is not exactly what they have in mind, no amount of cajoling is going to get you the job. As rabid Amiga fans, we get upset when someone tells us they can't use our work. Try not to let it bother you. You are competing against studios that are still using over fifty thousand dollars of IBM-based equipment to create results that are not much better than what we can do. Our strength in the market is that we produce good results for a fraction of what the other studios have to charge. Our overhead is less. Don't promise them the moon if you know you can't deliver.

We hear examples of people overstepping the boundaries every day. Recently we had the opportunity to bid on a job that needed full classical animation - the same sort of look and feel

that the Disney studio pioneered in the nineteen thirties. Three minutes of character animation was needed for the video and we carefully timed out just how long was required for us to get the desired results. In the end another studio won the bid by claiming they could get the same appearance at about a quarter of our price. More power to them if they could really pull it off. If the customer does request a certain style (let's say a ray traced HAM animation) show them examples (along with a quote) before you accept the job.



Less is more in computer graphics

Let's go back to the basics. You are hired to produce a short production, in this case a three item bar chart. Each bar grows in sequence. The client's logo is at the top of the screen and a couple of words are at the bottom. On the Amiga, such a simple assignment should take you no more than half an hour. In practice, it involves scanning in the logo with Digi-View in line art mode, doing a bit of clean-up and then drawing the graph with the grid on in DPaint. Sequence the picture with your favorite slideshow software, and then dump it to tape. Nothing could be easier, and the client would be pleased if that was all you did. Many studios, however, usually feel this overwhelming compulsion to doll up the graphics. Why not fill the bars with graduated color? Why not make the logo look like it's made of chrome? Because they didn't ask you to. Not only are you doing something uncalled for, you are presuming you know better than the people who hired you. That's not good business. What if you went to a car dealer and ordered a blue Chevrolet, but when you took delivery you received a yellow Ford? No extra charge of course, but you would feel like the salesperson was taking liberties with your opinion. It's in your own best interest to follow the assignment to the letter. That way there's less chance of a problem later on.

In business, the little extras cost more. You can gently suggest the chrome look if you think it will help, but you should also charge more for it. Don't be disappointed if your ideas are shot down. Know your client's needs - not everyone has the same taste as you. In our studio we use several portfolios. They range from a mundane slideshow of charts and graphs to a bizarre digital effects extravaganza. We show one to insurance companies, while the other is saved for producers of music videos. This system has worked out very well for us. Tailor your sales pitch to the needs of the customer. Your show reel may contain all the bells and whistles, while missing out on the simple motion required by the corporate market. If you've pulled out all the stops you may have a hard time convincing them that you can do the easy stuff as well.

Visual awareness

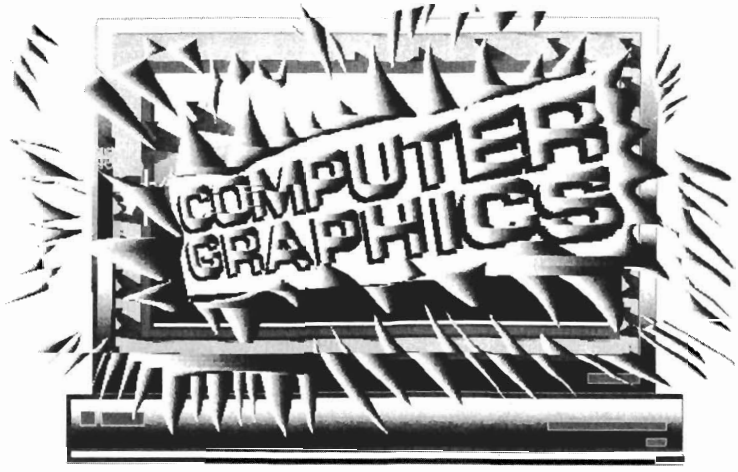
We have at our disposal literally hundreds of tools not available to the typical art director. We can twist, flip and warp our pictures in ways unheard of even five years ago. Though your new paint program may contain the neatest feature you have ever seen, you may never find a job where you can use it. Unfortunately, in many cases, the rest of the world has not yet become accustomed to these effects.

People today are more visually aware that at any time in history. We are all under constant media bombardment from every direction. We expect things to look a certain way and ignore those that don't. We may not understand why one commercial is more appealing to us than another, but as professionals we have to try our best to find out.

Many of the best tricks that we can do with our Amigas are not things that people are used to seeing every day. What looks wonderful to us may be just an annoyance to the viewer. Even more important, remember there is a time and place for everything. Digital video effects are used heavily in glossy shows like Entertainment Tonight. Corporate boardrooms are not the place for bizarre computer graphics... at least not yet. Don't let the gimmicks get in the way of the message.

The idea

The computer animator has to take more care when getting the message across. The movement has to accent the central idea, not get in the way. Let's go back to our bar chart for a moment. Instead of a still, the customer wants a little motion. We have three separate elements to deal with. The first is the company logo. They paid a lot of dollars to design this, so treat it with respect. The second is the title or subheading. Usually it's something along the line of 'Monthly Sales'. Third is the chart itself. The bars are going to grow to their current levels. Though this sounds pretty commonplace compared to some of the killer demos that you've seen, for the small studio this type of work is the bread and butter. Your assignment is to build interest in the fact that the customer's sales were X amount for the last month. The logo is important and establishes who is making the higher profits. The subhead gives the viewer necessary information about what is contained in the chart. It can be understated and should be there only to give details about



Using too many tricks loses the message

what is to come. The actual bars are the main point and should be treated with great significance. They are the central idea and everything else should just sit in the background and support them.

Think of every sequence as a short play. You are the director and set designer. You are also the production assistant and coffee maker. In other words, you control every aspect of the production.

Timing is crucial. Studies show that people will watch a plain still for about three seconds before their attention shifts. Fade in the logo at the top of the page. When it reaches its full intensity, hold it briefly and then bring on the next piece. You've established who is sponsoring the show; now for the data. In this case both the empty graph and the subheading text at the bottom should come on at the same time.

A chart without information in it serves no purpose. The animation of the growing bars should begin instantly, but once it starts it should be quite slow. This gives the sequence a little visual drama, and clients just love to watch the sales climb. Everyone pays full attention to something that is moving. When the first bar finishes, go immediately to the next in line. After all the bars are in place, hold the chart briefly and then move on to the next sequence.

Try to think of every production as wheels within wheels. Do comprehensive breakdowns. Look at the video as a whole, then divide that into individual sequences. Break the sequences down into single animations. Treating every piece as a small part of the whole will improve the flow of information.

Many times computer animators are guilty of information overload. Clients sometimes request too much data to appear on the screen at once. Gently try to steer them in other directions. The rule to live by is to try to keep no more than three separate elements on the screen at once. The viewer does not want to decipher any more than that. The 'rule of threes' avoids clutter on the screen.

Complex animation staging

Sometimes more is required than just moving blocks around. Hopefully, you'll get exciting work where you are required to create colourful graphics flying around the screen in complex maneuvers. Moving design follows the same general rules as static design. Too much movement happening at once can be dazzling but confusing.

The concept of staging for animation is called motion choreography. Every move must in some way be leading to the finished idea. As good stage acting must be thought through, so must good animation. Three dimensional animation brings even more into the picture. Forms have volume. Perspective comes into play. With 3D your are allowing the viewer to see into a universe that you have created. Let them see only what is needed to get the concept across. Working in three dimensions also requires a great deal of CPU time. The simpler the end results, the shorter the rendering time.

Here are some common motion choreography rules that you can apply to your own productions.

Set up the action. If you are introducing a new part to the production, give the audience a hint about what is coming next. If the picture is of a bare stage, then have a light come on before anything else happens. The audience knows that something is going to happen and they will pay attention. It's a signal that the show is about to start. Cues like this can be just as effective if done with sound. An orchestra tuning their instruments would accomplish the same thing as the light, in this instance.

Make sure that nothing interrupts or distracts from the flow of the sequence. If a character is moving on to center stage, get him there with as little fuss as you can. Skip any sight gags or other distractions at this point. If you can pop the character on to where he's supposed to be, so much the better.

Positioning of the graphics is critical. The rule here is to think in silhouettes. Will the picture read if you were to drop it down to black and white? Let's say you have a character that is picking up a shampoo bottle. He should hold it at arm's length to distract attention from himself and focus it where it belongs. Study commercials. On television you never see an actor hold a product in front of his body, even though that's the natural way of doing things. The same theory applies to text. If the drop shadow or perspective is getting in the way of the lettering, you should eliminate it.

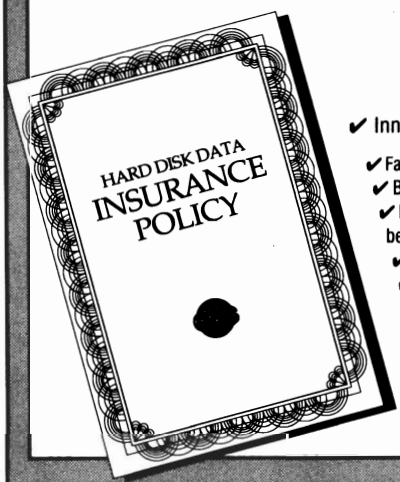
Our Amiga is one of the hottest new tools to be used in graphic design. It's a self contained audio-visual workstation. There are hundreds of programs available today that help present your message in thousands of ways. By sticking to basics you'll save time, keep the client happy and turn around the jobs faster. What could be simpler or more profitable?

FOR THE AMIGA

Protecting your valuable hard disk files is easier and faster than ever before!

QUARTERBACK™

*The **FASTEST** Hard Disk Backup Utility!*



Backup to or restore from:

✔ Floppy Disks ✔ Streaming tape (AmigaDOS-compatible)
✔ CLtd's Konica 10.7MB high-density floppy drive
✔ Inner-Connection's Bernoulli drive ✔ ANY AmigaDOS-compatible device

✔ Fast backup - 20MB in 30 minutes or less ✔ Uses two floppy drives (if available) for backup/restore with automatic switching ✔ Builds, sorts and displays catalog of files and subdirectories ✔ Provides FULL/Subdirectory/Individual file backup/restore ✔ Includes or excludes files by name (with wild cards), file date, or archive bit ✔ Calculates the number of floppies you'll need before you start ✔ Handles files of unlimited length, unlimited subdirectories and unlimited files per subdirectory ✔ Automatically formats diskettes with no delay as it writes ✔ Sequentially numbers and date/time stamps backup diskettes ✔ Checks the sequence number and date/time stamp of each diskette before restoring files from it ✔ Restores original date/time stamp, file notes, and protection bits on both files and subdirectories ✔ Runs with Workbench or CLI ✔ Produces backup/restore report to disk or printer ✔ Beeps for floppy change ✔ Accepts CLI parameters and batch command files ✔ Detects bad disks during backup or restore ✔ Convenient/user friendly error recovery ✔ Multitasking ✔ Runs in 512K ✔ No copy protection ✔ Works with all AmigaDOS compatible hard disk drives.

Only $69.95 Plus $3.00 shipping and handling, CO residents add appropriate sales tax.

Coming Soon!

OPTIMIZE HARD DISK PERFORMANCE

QUARTERBACK TOOLS™ provides a collection of high-quality, user friendly utilities to enhance use of your Amiga.

- ✔ Optimize hard disk organization to speed file access
- ✔ Undelete deleted files safely and easily
- ✔ Map disk space usage
- ✔ Verify readability of every block on disk
- ✔ Automatically map out bad sectors
- ✔ Detect and fix disk corrupted directories
- ✔ Verify file integrity
- ✔ Edit files and disk sectors
- ✔ Search for files throughout the disk
- ✔ Compare file contents

QUARTERBACK TOOLS™
works with any Amiga.

Read/Write Macintosh Disks

Mac-2-Dos™ transfers Macintosh files to and from the Amiga. You can quickly and easily import those graphic images from the Mac, or output your PostScript document to a Mac disk from which you can have it printed using high-quality, professional typesetting equipment. Mac-2-Dos reads, writes and formats single-sided disks (flat file format) or double-sided disks (hierarchical file format). It translates ASCII files and provides IFF conversion utilities to/from MacPaint and other popular Mac graphics formats.

Mac-2-Dos includes a custom interface and a Macintosh external 3.5-inch drive. Mac-2-Dos also includes a special driver which allows the Mac drive to read/write standard Amiga diskettes.

Mac-2-Dos will also be available without the Macintosh drive. This option DOES NOT include the ability to use the Mac drive to read/write Amiga diskettes.

DOS 2 DOS™

DOS-2-DOS transfers MS-DOS and Atari ST files to and from AmigaDOS!

DOS-2-DOS version 3.0 permits access to any MS-DOS volume available via AmigaDOS, including MS-DOS partitions on hard disks and MS-DOS volumes on LANS or SCSI networks.

✔ Supports single and double sided 5.25-inch as well as 3.5-inch 720KB MS-DOS diskettes ✔ Reads/Writes 3.5-inch Atari ST diskettes (GEM format) ✔ Reads a variety of 5.25-inch MS-DOS floppy formats via the CLTD Konica high-density floppy drive ✔ Converts ASCII file line-ending characters and translates extended ASCII character sets ✔ Supports full directory path names, with wild cards in the file names ✔ Allows selection of MS-DOS and AmigaDOS subdirectory and displays sorted directory listing ✔ Formats 3.5-inch and 5.25-inch MS-DOS diskettes and Atari ST diskettes ✔ Provides duplicate file name detection with query/replace options ✔ Provides TYPE and DELETE commands ✔ Permits renaming of files where file name restrictions occur ✔ Remains resident to permit AmigaDOS disk swapping.

Only $55.00
Plus $3.00 shipping and handling
CO residents add appropriate sales tax

CCS™

Central Coast Software
424 Vista Avenue, Golden, Colorado 80401 • Phone 303 / 526-1030 • FAX 303 / 526-0520
DEALER INQUIRES WELCOME

VISA

Assembly Language, Part 4

Allocating memory, and reading a file in one gulp

by Jim Butterfield
Copyright © 1989 Jim Butterfield

Jim Butterfield is a Toronto programmer and author whose articles, books, lectures and television programs have made his name a household word among Commodore users all over the world since the early days of the KIM-1 microcomputer.

In case you haven't got your 68000 documentation yet, the table opposite summarizes almost all the 68000 commands in reference form. Looking at the table, you can see the assembler opcode, the number of operands required, and what flags are likely to be affected. I've dropped a few of the 'dangerous' commands - those involving exceptions and privileged instructions. If you need them, you may look them up.

We've done quite a bit of working with data and files. As we did so, we were rather stingy with memory. A side effect of this stinginess was the elaborate use of registers; we had to map our A and D registers very closely to hold the maximum amount of information there.

This gives speed, of course; but there's a limit to how far you can go with this technique. It's time to open up memory a bit (no play on words intended) and show a more typical use of variables. You can't just help yourself to memory in a multitasking computer such as the Amiga; other processes might be using any given part of memory. We must 'apply' for the storage space we need.

We'll look a little more closely at the stack, which is policed by register A7; that's an easy place from which to obtain reasonably small quantities of working memory. And we'll ask for bigger chunks of memory from Exec by means of the AllocMem command.

Whether we take our memory from the stack or from the general area often called the 'heap', we must be careful to give back every piece of memory we borrowed. If we don't do so, memory might be forever lost from the Amiga, at least until the system is reset; and in the worst case, bad memory handling could trigger a visit to the black guru.

The project

Program *rfile* ('read file') will read a file and deliver it to the screen. Big deal: we did something more sophisticated than that last time; and for that matter, TYPE would do the same job. But we will approach the task in a slightly different way, and the program not only sets the stage for some sophisticated future file work, but also gives us a little usage bonus.

Our objective is to read the entire file in one shot, providing there is sufficient memory. This will be much faster than ten-characters-at-a-time, which our previous example did. It also opens up new possibilities. We won't go this far today, but consider: if an entire file is read into memory, we would be in good shape for such tasks as sorting or searching.

Again, our program will be run only from CLI. The command word will be followed by the name of the file, for example, *rfile s:startupII*. We will get to Workbench-started programs some day, but complex coding is involved; for the moment, we'll continue to refine our programming skills and system familiarity.

Mechanics

To read in the entire file, we must first of all get hold of enough memory to hold all the data. That will need a call to the Exec library subroutine AllocMem (Allocate Memory), with a later call to FreeMem (Free Memory) to give the memory space back. But how do we know how much memory to ask for? The file might be any size; we must ask for information on how big it is.

The best way to obtain a file's size is to ask for its FileInfoBlock (File Information Block), which contains many items telling about the file. Figure 1 shows a FileInfoBlock as it sits in memory; you will see that 124 bytes down, there's an item called 'File Size', giving the size in bytes. Just what we need.

Before we can get the FileInfoBlock, however, we must reserve a place for it in memory. All right: that's another call to

Quick 'ML 68000' Guide

INSTRUCTIONS (except privileged and exception types)

Mnemonic	Description	Operands	Flags affected
ABCD	Add Decimal	2	xzc
ADD	Add Binary	2	xnzvc
ADDA	Add Address	2	
ADDI	Add Immediate	2	xnzvc
ADDQ	Add Quick	2	xnzvc
ADDX	Add Extended	2	xnzvc
AND	Logical AND	2	nz
ANDI	Immediate AND	2	nz
ASL	Arithmetic Shift Left	1,2	xnzvc
ASR	Arithmetic Shift Right	1,2	xnzc
Bcc	Branch condition-code	1	
BCHG	Test/Change Bit	2	z
BCLR	Test/Clear Bit	2	z
BRA	Branch Always	1	
BSET	Test/Set Bit	2	z
BSR	Branch subroutine	1	
BTST	Test Bit	2	z
CLR	Clear (zero) operand	1	nz
CMP	Compare	2	nzvc
CMPA	Compare Address	2	nzvc
CMPI	Compare Immediate	2	nzvc
CMPM	Compare Memory	2	nzvc
DBCC	Test, Decr & Branch	2	
DIVS	Divide Signed	2	nzv
DIVU	Divide Unsigned	2	nzv
EOR	Logical Exclusive OR	2	nz
EORI	Immediate EOR	2	nz
EXG	Exchange Registers	2	
EXT	Sign Extend	1	nz
JMP	Jump	1	
JSR	Jump to Subroutine	1	
LEA	Load Effective Address	2	
LINK	Link, Allocate stack	2	
LSL	Logical Shift Left	1,2	xnzc
LSR	Logical Shift Right	1,2	xnzc
MOVE	Move Data	2	nz
MOVEA	Move Address	2	
MOVEM	Move Multiple Registers	2	
MOVEP	Move Peripheral Data	2	
MOVEQ	Move Quick	2	nz
MULS	Multiply Signed	2	nz
MULU	Multiply Unsigned	2	nz
NBCD	Negate Decimal	1	xzc
NEG	Negate	1	xnzvc
NEGX	Negate Extended	1	xnzvc
NOP	No Operation		
NOT	Logical Complement	1	nz
OR	Logical Inclusive OR	2	nz
ORI	Immediate OR	2	nz
PEA	Push Effective Address	1	
ROL	Logical Rotate Left	1,2	nzc
ROR	Logical Rotate Right	1,2	nzc
ROXL	ROL with Extend	1,2	xnzc
ROXR	ROR with Extend	1,2	xnzc
RTR	Return, Restore CC		
RTS	Return from Subroutine		
SBCD	Subtract Decimal	2	xzc
Scc	Set Byte on Condition	1	
SUB	Subtract Binary	2	xnzvc
SUBA	Subtract Address	2	
SUBI	Subtract Immediate	2	xnzvc
SUBQ	Subtract Quick	2	xnzvc
SUBX	Subtract Extended	2	xnzvc
SWAP	Swap Register Halves	1	nz
TAS	Test Byte and Set Bit	1	nz
TST	Test Operand Z/N	1	nz
UNLK	Unlink	1	

Popular Addressing Modes

Dn	Data Register Direct
An	Address Register Direct
(An)	Address Register Indirect (ARI)
(An)+	ARI with post-increment
-(An)	ARI with pre-decrement
$d(An)	ARI with displacement
$d(An,Dm)	ARI with index
$aaa	Absolute (short or long)
$d(PC)	Program Counter with displacement
#$xxxx	Immediate

Condition Codes (u/s = unsigned/signed)

T	True		always
F	False		never
HI	High	u	not-C and not-Z
LS	Low/Same	u	C or Z
CC	Carry Clear	(u, =HS)	not-C
CS	Carry Set	(u, =LO)	C
NE	Not Equal		not-Z
EQ	Equal		Z
VC	Overflow Clear		not-V
VS	Overflow Set		V
PL	Plus		not-N
MI	Minus		N
GE	Greater/Equal	s	N eor V
LT	Less Than	s	not(N eor V)
GT	Greater Than	s	not Z & (N eor V)
LE	Less/Equal	s	Z or not (N eor V)

Chart compiled by Jim Butterfield

AllocMem for the 260 bytes, with a later release via FreeMem. No problem on that part.

We get a FileInfoBlock by calling the DOS library subroutine Examine. We see that this routine wants to be given the memory location at which it can place the FileInfoBlock. That's fine: we know how to get that chunk of memory. But it also wants a thing called the 'lock' for the file. How do we get that?

We get a file lock by calling DOS subroutine Lock. We call this a 'read' lock (you might like to think of it as an 'interlock'). This means that other programs could read this file at the same time that we are reading it; but nobody may write to the file until we're finished reading.

That's about it; but it would be worth while going through the whole procedure in its final order.

- **Lock**: to get a read-lock on the file;
- **AllocMem**: get a block of memory to hold the FileInfoBlock;
- **Examine**: read in the FileInfoBlock for the file;
- **AllocMem**: get memory sufficient to hold the entire file;
- **Open**, **Read**, and **Close**: read the whole file;
- **FreeMem**: return the whole-file block of memory;
- **FreeMem**: return the FileInfoBlock piece of memory;
- **UnLock**: release the file.

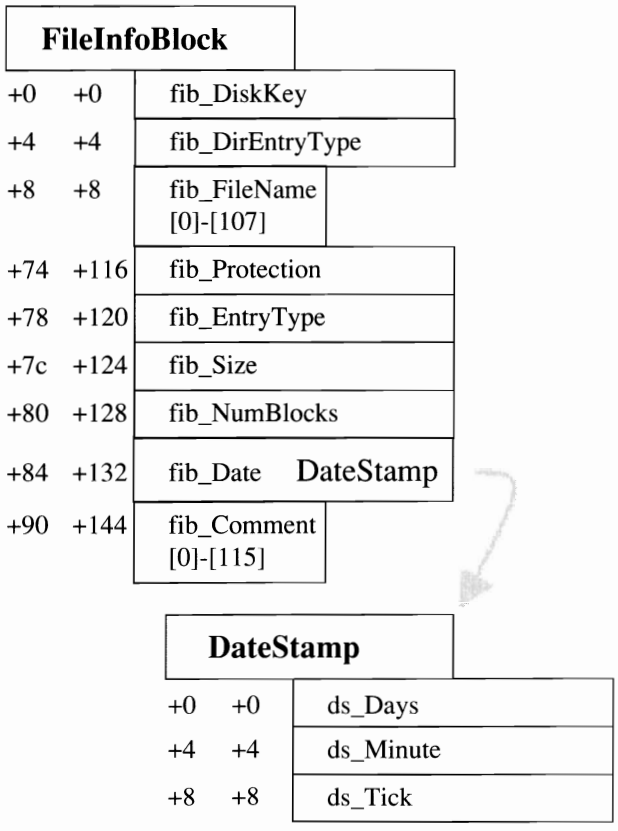


*Figure 1: the **FileInfoBlock** structure*

That's quite a sequence, but it's logical. You can see how the items 'nest', so that the top and bottom lines match each other, and so on.

Background information

We'll use part of the stack for our working variables. The amount of memory will be set aside using the LINK command. At the end of the program, we must be careful to return this memory with the UNLK command.

Figure 2 shows the stack that we grab using command LINK, and the use we make of this area. This time, we will simply use the offsets for the part of memory we have reserved for this use. In the future, we will get more 'meaningful' source code by using the EQU ('equate') command to give symbolic names to these data units.

Note that there's a parallel operation involved here. Commands LINK and UNLK handle allocating memory from the stack. Subroutines AllocMem and FreeMem do the same kind of thing, except that the allocation is from general memory. In both cases, we must take care to give back the memory we have borrowed. There are differences in usage of these two operations. The stack may have only a few thousand bytes, so we would not want to grab a huge amount of memory from there. LINK and UNLK are quick and easy commands, so for a small amount of memory, they are just what is wanted.

AllocMem takes more setting up, but offers several advantages. We can ask for a lot of memory, if we wish. We can ask for a special kind of memory (chip or fast memory, for example). And one more thing: we will be given memory that is *long-word aligned*. We'll talk about this in a moment; this alignment is handy for receiving a FileInfoBlock.

The FileInfoBlock structure

A *structure* is a part of memory organized in some particular way. The FileInfoBlock structure is a piece of memory organized to hold information about a file or directory. You'll see a 'memory map' of this kind of structure in Figure 2. You can get details of a structure from your *include* files, if you have them. The FileInfoBlock will be found in file *dos.h*. For the moment, we'll use the actual offsets.

We don't know where this structure will be until we have allocated memory to hold it. But that's easy to work out in the program. A register will hold the address of the allocated memory, and we'll use the address in *indirect* mode to get at the block itself. More precisely, we can reach any part of the structure that we want with an 'offset'; our addressing mode will be 'Indirect with Offset'.

A first glimpse of BPTRs (B-Pointers)

The FileInfoBlock is fetched for us by the part of Amiga's operating system called DOS (for Disk Operating System). DOS

was written in a language called BCPL. You don't need to know this language, but it has an interesting side-effect: it likes addresses that divide evenly by four.

In fact, the DOS system often likes to work with addresses that have been divided by four. Such addresses are called B-Pointers (BPTRs). You will meet them as you get deeper into DOS system mysteries, such as Workbench startup code.

At that time, you might get a BPTR and need to multiply it by four to get the 'real' address. Not hard, and some people argue that it will be an advantage when we move along to the 68030 processor chip, but it's one more thing to keep in mind.

For the moment, the effect is this: if we ask for a FileInfoBlock, we must supply a location to hold this block that is on a longword boundary. In other words, the FileInfoBlock must start at an address that divides evenly by four.

No problem. AllocMem always gives you this kind of address ('longword-aligned'). That's why we use it for the relatively small FileInfoBlock. It would seem easy to just grab 260 bytes from the stack; but if we did that, we could not be sure that the first address would divide exactly by four.

Coding start

Use your favourite editor to create a new file called *rfile.asm*. We'll talk as we go through the code.

First things first. A title followed by a few xref lines: some of these we have used before, and some are new.

```
;
;   Load and Print file - Jim Butterfield.

; Exec library calls
   xref _LVOAllocMem     ;-$C6
   xref _LVOFreeMem      ;-$D2
   xref _LVOCloseLibrary;-$19E
   xref _LVOOpenLibrary ;-$228

; DOS library calls
   xref _LVOOpen         ;-$1E
   xref _LVOClose        ;-$24
   xref _LVORead         ;-$2A
   xref _LVOWrite        ;-$30
   xref _LVOOutput       ;-$3C
   xref _LVOLock         ;-$54
   xref _LVOUnLock       ;-$5A
   xref _LVOExamine      ;-$66
```

The following numbers are needed by specific routines. We will make them symbols, in the hope that the routines will be easier to read. ACCESS_READ will be used by Lock, and MODE_OLDFILE is needed by Open. INFOSIZE is the size of a FileInfoBlock, and SIZEPOINTER is the place within the block that we will find the size of the file we are investigating.

SP before link →	Old value of a4	←New a4 value
-$4(a4)	Pointer to command tail	
-$8(a4)	Length of command tail	
-$c(a4)	Dosbase library pointer	
-$10(a4)	Output handle	
-$14(a4)	File read lock	
-$18(a4)	Allocated FileInfoBlock	
-$1c(a4)	File size	
-$20(a4)	Allocated file date area	
SP after link →		

*Figure 2: Stack frame variables in the **rfile** program*

```
;
ACCESS_READ   equ  -2
MODE_OLDFILE  equ  1005
INFOSIZE      equ  260
SIZEPOINTER   equ  124
```

Now: we're going to clear out some register space and put working values into memory. We will need 8 longwords - 32 bytes - for the values that are planned. These values might be shown in a table as follows:

-$20 File buffer address
-$1C File buffer size
-$18 FileInfoBlock address
-$14 File 'lock'
-$10 Output Handle
-$C Dos Library pointer, 'DosBase'
-$8 Length of 'command tail'
-$4 Address of 'command tail'

Don't worry that the table numbering consists of negative values. We'll cover that in a moment.

None of the items shown above have any values yet. We'll fill them in as we go. In fact, these items haven't even got a location yet. Let's ask the stack to give us 32 bytes to hold these numbers.

Such items usually carry symbolic names, so that (with the right EQUates) we would say **OutHandle(a4)** instead of **-$10(A4)**. It looks much better that way. This one time, however, I'm going to give the raw numbers, in the hope that this will help you see the mechanism at work.

```
;-- Initial setup:
Startup link a4,#-$20
```

The stack has given up 32 (hex 20) bytes; register A4 points just above the top of this area. The highest longword (a long

word is four bytes long, of course) would go from 4 bytes below the address in A4 to the byte immediately below A4.

If we put brackets around A4, we create an 'indirect address' - the address to which A4 points. If we tack a value just before this, for example -$4(A4), the value acts as a 'displacement' that adjusts the indirect address upward or downward. For example, if A4 held the address $2468, the expression -$4(A4) would refer to the contents of address $2464.

Can we be sure there is enough space on the stack? In the case of 32 bytes, there is little danger. For bigger chunks of the stack, we might need to do some sophisticated testing.

```
move.l  a0,-$4(a4)      ; Address, Command Tail
move.l  d0,-$8(a4)      ; Length, Command Tail
```

The contents of A0 and D0 are safely stored away. We can do our usual job of opening the Dos library.

```
lea     dosname(pc),a1  ; Name 'dos.library'.
clr.l   d0              ;   Any version (0)
move.l  $4,a6           ;   Using Exec library
jsr     _LVOOpenLibrary(a6) ;  Open Dos library.
move.l  d0,-$C(a4)      ; Remember DosBase ptr.
beq.s   StartupQuit     ;       DOS not opened!
```

Note that the **MOVE.L D0** instruction sets the zero condition flag, which is tested by the following **BEQ**. If we get to this point without trouble, we may call DOSinit.

```
bsr.s   DOSinit
```

When DOSinit returns, we have some cleaning up to do: closing DOS and returning ('unlinking') the memory borrowed from the stack.

```
            move.l  -$C(a4),a1 ; DosBase to a1;
            move.l  $4,a6      ;     then using
                               ;     Exec library,
            jsr     _LVOCloseLibrary(a6)
                               ;  close Dos library.
StartupQuit unlk    a4
            rts                ; End of Program
```

Phase 1 was sucessful, and we have reached DOSinit. We do the usual job of obtaining the Output handle, storing it in our stack table. As usual, we stuff a zero behind our file name, and strip out any spaces at the beginning.

```
;
;-- Get CLI outhandle:
DOSinit     move.l  -$C(a4),a6    ; set Dos library
            jsr     _LVOOutput(a6); get CLI outhandle,
            move.l  d0,-$10(a4)   ; & then remember it.
;
; -- Change arg file name to C type of string:
checklen    move.l  -$4(a4),a3    ;command tail address
            move.l  -$8(a4),d3       ; command tail length
            move.b  #0,-$1(a3,d3.W) ; binary zero at end
skipspc     cmp.b   #$20,(a3)+
            beq.s   skipspc
            subq.l  #1,a3
            move.l  a3,-$4(a4)    ; save corrected address
```

Now for the new stuff. The DOS manual shows the format of command Lock to be:

```
lock = Lock(name,accessMode)
        D0   D1     D2
```

For a read-lock, our access mode will be a value of -2. In memory, this would be hexadecimal FFFFFFFE.

```
; get a 'lock' on the file, if possible.
     move.l  a3,d1
     moveq   #ACCESS_READ,D2 ; get a read lock
     jsr     _LVOLock(a6)
     move.l  d0,-$14(a4)     ; store the lock pointer
     beq     DosQuit         ; couldn't get lock
```

There are unlikely reasons ("the file is currently being written") why we would not get a lock on the file. There is a likely reason: you have spelled the file name wrongly, or have given the wrong path. To save space and typing, we won't send an error message if any such thing happens. We'll just give up.

If we do get a lock, we're ready to go for the file information block. Let's ask for that 260 bytes of memory. We must switch libraries here, since AllocMem is an Exec function.

The ROM Kernel Manual gives the format for AllocMem as:

```
MemoryBlock = AllocMem(byteSize, requirements)
     D0                   D0          D1

; ask for memory to hold a FileInfoBlock.
     move.l  $4,a6           ; this calls for Exec
     moveq   #0,d1           ; any type of memory
     move.l  #INFOSIZE,d0    ; size of block needed
     jsr     _LVOAllocMem(a6); go get it
     move.l  d0,-$18(a4)     ; save the block address
     beq.s   DropLock        ; quit if no memory
```

Your Amiga would be truly crammed if you couldn't find a measly 260 bytes of memory; but the code to check is there in any case. The type of memory was specified as zero: anything, chip or fast memory. Now we switch back to the DOS library for Examine.

The AmigaDos Manual shows the format of Examine as:

```
success = Examine(lock,FileInfoBlock)
  D0              D1      D2

; Now get information on the file
```

```
        move.l  -$C(a4),a6        ; set Dos library
        move.l  -$14(a4),d1       ; lock
        move.l  d0,d2             ;file info block address
        jsr     _LVOExamine(a6)   ; go for the data
; Assume Examine never fails.  Go for the size.
        move.l  d2,a2             ;file info block address
        move.l  SIZEPOINTER(a2),-$1C(a4)   ; file size
        beq.s   FreeInfo
```

Once you have a good lock, it's hard to imagine Examine failing, although the documentation shows that such a signal is possible. You could have a different problem, though: the file might have a size of zero. This could be an 'empty' file item, but it's more likely that we have gotten a lock on a directory rather than a file.

We have established a file size. We are ready to go back to Exec to ask for another AllocMem. We need to do careful bookkeeping, since there are blocks to be given back and a file to be unlocked.

```
; OK, we know the size.  Ask for a block that big
        move.l  $4,a6             ; this calls for Exec
        moveq   #0,d1             ; any type of memory
        move.l  -$1C(a4),d0       ; size of block needed
        jsr     _LVOAllocMem(a6); go get it
        move.l  d0,-$20(a4)       ; save the block address
        beq.s   FreeInfo          ; quit if no memory
```

It's quite possible for us to have found a big file, too big for the memory we have available. If so, we can't do the job and the program gives up.

Now, back to DOS and we will read the entire file.

```
; Whew!  Now ready to read the file.
        move.l  -$C(a4),a6 ; set Dos library
        move.l  -$4(a4),d1 ; filename pointer
        move.l  #1005,d2   ;  MODE_OLDFILE (for reading)
        jsr     _LVOOpen(A6)
        move.l  d0,d6      ; file inhandle
        beq.s   FreeFileMem   ; no good, quit
; Read whole file in one shot.
        move.l  d6,d1          ; input file handle
        move.l  -$20(a4),d2    ; input buffer address
        move.l  -$1C(a4),d3    ; input buffer size
        jsr     _LVORead(A6)   ; so read it
        move.l  d6,d1          ; use the handle..
        jsr     _LVOClose(A6)  ; to close the file
```

Open, Read, and Close. The file has done its job and all the data is now in memory.

```
    bsr.s       MainJob       ; the main job
```

The main job has been done. Now to clean up the two memory blocks and the file lock. I hope you notice how the stack 'frame', indexed from A4, makes the bookkeeping easier.

The ROM Kernel Manual gives the format for FreeMem as:

```
AllocMem(MemoryBlock,byteSize)
             A1          D0

FreeFileMem  move.l  $4,a6          ; set Exec library
             move.l  -$20(a4),a1
                                    ; memory block address
             move.l  -$1C(a4),d0    ; size of block
             jsr     _LVOFreeMem(a6)
                                    ; release this memory
FreeInfo     move.l  $4,a6          ; set Exec library
             move.l  -$18(a4),a1
                                    ; memory block address
             move.l  #INFOSIZE,d0   ; size of block
             jsr     _LVOFreeMem(a6)
                                    ; release this memory
```

The DOS manual shows the format of command UnLock to be:

```
UnLock(lock)
        D1

DropLock   move.l  -$C(a4),a6       ;set Dos Library
           move.l  -$14(a4),d1      ;take off the lock
           jsr     _LVOUnLock(a6)
DosQuit    rts                      ;exit program.
```

Our main job is something of an anticlimax. At this point we might be sorting, indexing, or doing other wonderful things. Instead, we simply output the whole file.

One Write command sends the whole file. I hope your selected file contains text, because Control-C won't stop you this time.

```
;
; The file is in the buffer.  Write it in one shot!
; Register A6 contains DOSBase
MainJob move.l  -$10(a4),d1      ;file handle
        move.l  -$20(a4),d2      ;outbuffer
        move.l  -$1C(a4),d3      ;length
        jsr     _LVOWrite(A6)
        rts
dosname dc.b    'dos.library',0
   end
```

The end product

Assemble and link the program, and you'll find that *rfile* will type a file quickly to the screen. Nothing new, it seems, but you have had a chance to examine some new aspects of programming and file handling.

There's an extra wrinkle, however: file copying. Even if a file is not 'printable', we can send copies of it anywhere we like by using redirection. Want to copy a file called *dog* to *df1:cat*? The Copy command does the job, of course, but you could also try: **rfile >df1:cat dog**. It will do the job faster than Copy,

if you have sufficient RAM to hold the file. And it's interesting to know you can do the job in a program of only a few hundred bytes.

Recap: program *rfile*

Here's the whole program in one piece, without any text interspersed:

```
;
;   Load and Print file - Jim Butterfield.

; Exec library calls
            xref _LVOAllocMem          ;-$C6
            xref _LVOFreeMem           ;-$D2
            xref _LVOCloseLibrary      ;-$19E
            xref _LVOOpenLibrary       ;-$228
; DOS library calls
            xref _LVOOpen              ;-$1E
            xref _LVOClose             ;-$24
            xref _LVORead              ;-$2A
            xref _LVOWrite             ;-$30
            xref _LVOOutput            ;-$3C
            xref _LVOLock              ;-$54
            xref _LVOUnLock            ;-$5A
            xref _LVOExamine           ;-$66
;
ACCESS_READ            equ -2
MODE_OLDFILE           equ 1005
INFOSIZE               equ 260
SIZEPOINTER            equ 124

\s-$20 .. File buffer address
\s-$1C .. File buffer size
\s-$18 .. FileInfoBlock address
\s-$14 .. File "lock"
\s-$10 .. Output Handle
\s-$C ... Dos Library pointer, "DosBase"
\s-$8 ... Length of "command tail"
\s-$4 ... Address of "command tail"

;-- Initial setup:
Startup     link   a4,#-$20
            move.l a0,-$4(a4)         ; Address, Command Tail
            move.l d0,-$8(a4)         ; Length, Command Tail
            lea    dosname(pc),a1     ; Name 'dos.library'.
            clr.l  d0                 ;   Any version (0)
            move.l $4,a6              ;   Using Exec library
            jsr    _LVOOpenLibrary(a6)  ;   Open Dos library.
            move.l d0,-$C(a4)         ; Remember DosBase ptr.
            beq.s  StartupQuit        ;      DOS not opened!
            move.l -$C(a4),a1         ; DosBase to a1;
            move.l $4,a6              ;   then using Exec library,
            jsr    _LVOCloseLibrary(a6)    ;   close Dos library.
StartupQuit unlk   a4
            rts                       ; End of Program
;
;-- Get CLI outhandle:
DOSinit     move.l -$C(a4),a6         ; set Dos library
            jsr    _LVOOutput(a6)     ;   get CLI outhandle,
            move.l d0,-$10(a4)        ;   & then remember it.
;
; -- Change arg file name to C type of string:
checklen    move.l -$4(a4),a3         ; command tail address
            move.l -$8(a4),d3         ; command tail length
            move.b #0,-$1(a3,d3.W)    ; binary zero at end
skipspc     cmp.b  #$20,(a3)+
            beq.s  skipspc
            subq.l #1,a3
            move.l a3,-$4(a4)         ; save corrected address
            move.l a3,d1
            moveq  #ACCESS_READ,D2    ; get a read lock
            jsr _LVOLock(a6)
            move.l d0,-$14(a4)        ; store the lock pointer
            beq    DosQuit            ; couldn't get lock
            move.l $4,a6              ; this calls for Exec
            moveq  #0,d1              ; any type of memory
            move.l #INFOSIZE,d0       ; size of block needed
            jsr    _LVOAllocMem(a6)   ; go get it
            move.l d0,-$18(a4)        ; save the block address
            beq.s  DropLock           ; quit if no memory
; Now get information on the file
            move.l -$C(a4),a6         ; set Dos library
            move.l -$14(a4),d1        ; lock
            move.l d0,d2              ; file info block address
            jsr _LVOExamine(a6)       ; go for the data
; Assume Examine never fails.  Go for the size.
            move.l d2,a2              ; file info block address
            move.l SIZEPOINTER(a2),-$1C(a4)  ;file size
            beq.s  FreeInfo
; OK, we know the size.  Ask for a block that big
            move.l $4,a6              ; this calls for Exec
            moveq  #0,d1              ; any type of memory
            move.l -$1C(a4),d0        ; size of block needed
            jsr    _LVOAllocMem(a6)   ; go get it
            move.l d0,-$20(a4)        ; save the block address
            beq.s  FreeInfo           ; quit if no memory
; Whew!  Now ready to read the file.
            move.l -$C(a4),a6         ; set Dos library
            move.l -$4(a4),d1         ; filename pointer
            move.l #1005,d2           ; MODE_OLDFILE (for reading)
            jsr    _LVOOpen(A6)
            move.l d0,d6              ; file inhandle
            beq.s  FreeFileMem        ; no good, quit
; Read whole file in one shot.
            move.l d6,d1              ; input file handle
            move.l -$20(a4),d2        ; input buffer address
            move.l -$1C(a4),d3        ; input buffer size
            jsr    _LVORead(A6)       ; so read it
            move.l d6,d1              ; use the handle..
            jsr    _LVOClose(A6)      ; to close the file
            bsr.s  MainJob            ; the main job
FreeFileMem move.l $4,a6              ; set Exec library
            move.l -$20(a4),a1        ; memory block address
            move.l -$1C(a4),d0        ; size of block
            jsr    _LVOFreeMem(a6)    ; release this memory
FreeInfo    move.l $4,a6              ; set Exec library
            move.l -$18(a4),a1        ; memory block address
            move.l #INFOSIZE,d0       ; size of block
            jsr    _LVOFreeMem(a6)    ; release this memory
DropLock    move.l -$C(a4),a6         ; set Dos Library
            move.l -$14(a4),d1        ; take off the lock
            jsr    _LVOUnLock(a6)
DosQuit     rts                       ;   exit program.
;
; The file is in the buffer.  Write it in one shot!
; Register A6 contains DOSBase
MainJob     move.l -$10(a4),d1        ;file handle
            move.l -$20(a4),d2        ;outbuffer
            move.l -$1C(a4),d3        ;length
            jsr    _LVOWrite(A6)
            rts
dosname     dc.b   'dos.library',0
   end
```

Colour Video Displays

Where do we go from here?

by Ben Blish

Ben Blish is president and founder of Black Belt Systems, a Montana-based Amiga software development company. Among the Amiga products he has developed are the electronic-CAD programs PCLO and PCLOplus. He is currently involved in a number of hardware-oriented projects for the Amiga.

Today, the Amiga boasts what is arguably one of the most colourful and flexible video displays available in a stock microcomputer system. In fact, its capabilities still outstrip many of the expensive optional graphics adapters for the mainstream segmented architecture micros.

The Amiga has been the victim of continual 'snubs' and misinformation in the media - the most recent example of this stated, in a popular magazine, that the Amiga had a palette of 'only' 4096 colours, compared to the 'much larger' Mac and PS/2 display systems. In fact, the Mac and PS/2 can display far fewer colours simultaneously than the Amiga. They do 256 colours at one time, from a palette that is much larger - 16 million colours - than the Amiga's available colour range. The key here is that the Amiga's 4096 colours may or may not be from a palette. HAM (Hold And Modify) mode, originally thought to be more of a curiosity than a practical display mode, has proven itself conclusively as *far* superior to palette-driven colour display modes for the display of real-world type images such as faces, meals - virtually anything with a great variety of colours.

The thrust of this article is to define what a palette is and contrast it to non-palette colour systems, to discuss how many colours is enough (an issue far too often treated by the application of reams of quotes from learned individuals on just how many colours you can see), and to try and point out some of the issues that are affected by these choices. From this, I will draw some conclusions, such as where bit-planes are appropriate, where pixel-planes are appropriate, and why the issue needs to be dealt with right now. A great deal of the initial material is simple, and is presented so that issues you have probably already been exposed to are fresh in your mind. It's not a complete basic introduction, but we'll cover most of the main points. As the article proceeds, I hope that you'll be jogged into thinking along the same lines by the initial material. Unfortunately, the issues treated here would require volumes to cover in fine detail, so we'll wing it a bit.

Colour generation in computer video

Computer video displays generally create colour on a computer monitor by providing three control signals to the monitor. Each signal controls the brightness of a specific colour - Red, Green, and Blue. This is where the term RGB comes from: it's the first letter of each of these colours. In a simple system, the three signals have only two states - on and off. That gives you the following possibilities on the monitor's display:

Red	Green	Blue	
off	off	off	**Black**
off	off	on	**Blue**
off	on	off	**Green**
off	on	on	**Aqua** (Cyan)
on	off	off	**Red**
on	off	on	**Purple** (Magenta)
on	on	off	**Yellow**
on	on	on	**White**

This is generally referred to as an eight-colour display, even though it really only provides six colours - in common usage, neither black or white is called a colour (although, technically, white can be considered one). If you want a display where each pixel on the display can be one of these eight possibilities, you need three memory bits per pixel: each bit representing the on or off state of one of the Red, Green, and Blue signals. In a 640 by 400 display (such as the Amiga's hi-res interlace display) you will need 640 x 400 x 3 bits of memory to create such a display - that's 768 thousand bits, or 96 thousand bytes.

If you want more colours, you need to add more bits. Two bits, for instance, could provide on, off, 1/3 on, and 2/3 on levels for a particular signal. If each signal has this capability, the same calculations reveal that you now need 188 thousand bytes to represent the image. Sound like a lot? It is - and the more memory the display requires, the more difficult it is to design and, of course, the more costly it gets.

Palettes

The colours in our eight-colour example, while bright and attention getting, are not particularly useful if you need to do

any kind of subtle display change: shadows that highlight an icon, for instance, or any kind of smooth transition from one colour to another. To address this need without a larger cost in memory, the display palette was created. Sticking with our three bits: This is extremely similar to the artist's palette, where the artist holds a tray with eight pots of colour that can be used at any one time on the current creation; while there are only eight paint pots, they can be adjusted (by adding other paints from a larger selection nearby) to represent other colours. The analogy is almost exact, and fails only when you consider that after a while, the artist will end up with pots that look like mud, whereas a computer's palette always mixes its colours fresh and so is more flexible in that regard.

In an eight-colour display, you have the choice of using any one of the eight at any time. No more, no less. A palette again brings up the issue of bits of memory versus number of colours available, but the math is astonishingly different. If we want to be able to have 4096 colours available (again, as is the case with the current generation of Amiga hardware), you need four bits per colour signal. These four bits provide fifteen separate luma (brightness) levels, plus no luma (black). You get the total number of colours by multiplying the three signal possibilities against each other - 16 x 16 x 16 = 4096. That works out to twelve bits of precision per pixel. But since you are choosing colours from a group of eight, the memory requirement is only 8 x 12 bits, completely negligible in a hardware design. Using a palette in this manner provides a huge increase in display quality and loses you nothing as compared to an eight-bit display without a palette. After all, you can always set the twelve-bit palette to values that would provide the same colours that you would have had without it.

This establishes an important concept: having a palette is better than not having a palette. There are some 'tricks' that can be done with a palette that are difficult or impossible without one, too. Amiga owners have all seen the result of colour cycling; you just can't do that without a palette. Palette availability is the key to a number of simple animation techniques you often see in places like your television weather map. To this I would add only that the palette needs to be larger than or equal to the number of colours choices available, or it can actually hurt the display capability. This becomes important when you move out of what I (cynically) call the crayon hardware range into displays that can simultaneously display 4096 or more colours.

How many colours is enough?

Let's step back from the arguments of the physiological experts, the psychologists, and those who would have you buy a 256-colour 'paletized' system (not me!). Take your paint program. Most of these will create a colour range for you: create the brightest red you can at one end of a sixteen-colour range, and black at the other end. Use a spread function to create the 14 levels of red between black and full red intensity, or do it manually. That is absolutely the best that a 4096 colour system can do to give you a smooth brightness change on one particular colour. Simply put, you can see the steps between the colours, and that tells you immediately that they are not fine enough.

The objective is for the computer to just barely exceed your ability to see a change while still executing one on the display. This way, you can smoothly and clearly create any nuance of colour you might require for *any* conceivable image. But you will note I said that this is the *best* that a 4096 colour system can do - it can do far worse. Consider an orange colour, consisting of red, green at half the intensity of the red, and no blue. You can have seven levels of this colour in a 4096 colour system, and you can certainly see steps in that! Here are the levels:

Red	Green
14	7
12	6
10	5
8	4
6	3
4	2
2	1

Any other combination will be close - but the ratios of the colours are different, and so you are not seeing exactly the same hue. It gets worse yet. What about a colour with full red, half blue, and 1/16th green? You get - count them - one level. There are many colours you can't represent at all in such a system, and in every case steps between individual colours are best described as coarse.

I have a 256 thousand colour display sitting on my lab bench where I work, with an Amiga about three feet away (underneath it). My company has a hardware device that handles images to 256 thousand colours, and this is what the display is used to view. I can view the images on either this display or the Amiga's HAM screen. In short, I can tell you that, (1) the 256 thousand colour display, while it actually has less resolution (it's 256 x 240) looks far better than the Amiga's 4096 colour one (using HAM mode); and (2) I can still see colour steps there - and easily. A 256 thousand colour display has six bits per gun, for 63 levels per signal plus black. Instead of doing the math on this display, let me just tell you for the moment that it is not precise enough to meet the stated goal of "finer steps than you can see", and proceed to a 24-bit display: sixteen million simultaneous colours on screen at once. Whew! Sounds impressive enough, and one is tempted to remark that it's 'obviously' too much. Let's look at it carefully then, from a practical standpoint - just as we did the 4096 colour display.

A 24-bit system provides eight bits per signal, for 255 brightness levels and black. 256 x 256 x 256 = 16 million plus, and that's where the description comes from. Using the orange example as we did above, you will find that there are 128 possible shades of orange available - not bad at all, and in fact this is better than you can see. Great. Even if you take the red, 50% blue, 1/16th green, you get sixteen accurate brightness levels

to play with of this exact colour. Sixteen levels, as we showed by example previously, isn't a whole lot, and you can certainly see it. On the other hand, with many colours available that are just a few millionths away from accurate, you can 'fudge' it if need be and get away with it in the sense that you won't be able to tell, by eye, that any fudging was done. This has been repeatably demonstrated in many experiments, and so I'll just pass that on now, without delving into yet another area of whys and wherefores.

I would point out, though, that when a program has to generate a smooth set of colour transitions, it's relatively easy, with integer (fast) math, to create an accurate linear range. When you begin to do fudge calculations, you need to take into account the various differences in red, green, and blue colour sensitivities of the human eye, and the calculations slow down if you keep the best possible approximation on screen. If don't do that, fudged colour changes become visible much sooner than they would otherwise. One thing that I always argue for is to make the hardware do all it can - it frees the processor for continually higher level operations, given you more bang for your time, which, in my opinion, is the single most valuable commodity we have. Non-renewable, elective, and something few of us - companies or individuals - have enough of.

It so happens that 24 bits is an extremely convenient colour selection for a number of other reasons important to users of computers such as engineers, scientists, statisticians, and so on. That same 16 million colour range provides 256 colours with 256 available levels - including white, incarnated as grey levels. Since the smallest natural data item a 68xxx processor can handle is an 8-bit word, which just 'happens' to be the number of bits needed to represent 256 combinations, it turns out that a display like this is appropriate for in-place computations on data. 256 levels of any one colour is more than you can see - in fact, 256 levels of grey is more than you can see, given that the dilation of your eye is constant (which is usually accepted to be the case when pondering a video display, unlike night adaptation. The change in your eye's dilation, while it does extend the range of brightness you can perceive in a practical sense, has moved your vision's limited dynamic range elsewhere - not truly extended it in the linear sense).

This allows data to be maintained in the display (which saves memory, sort of), calculated in the display conveniently (because of the size of the data word), quickly (because you didn't have to move it from somewhere else) and usefully (because the data set can extend beyond your ability to perceive its limits). To this you add those 256 sets of colours, all with this dynamic range, and you now have multiple representations of data sets with unique visual identities (by colour) and unique values (by brightness).

It even gets better. 24 bits fits very nicely inside a 32-bit word, leaving eight bits free. With all this discussion about computational efficiency, do you think I would leave them alone? No chance. We'll look at them a little later - and I think you'll be interested in what I propose.

We can also take comfort in the fact that the 'experts', by and large, agree that 24 bits is more than enough. It's kind of amusing that few of them agree on why this is so, but at least they concur on that main point.

The usefulness of palettes

When you look at an Amiga HAM image, it should be clear to you immediately that having 4096 colours available all at once does wonderful things for the *realism* of an image. The shadows cast by a translucent orange object on green felt illuminated from above are not representable with eight colours, one of which is black. From this it seems to follow that more colours are better, when you want to represent a real image - or one with finesse, if you will. That's true, essentially. But it costs you - in memory, in engineering talent, and in processing time. You see, the computer has to handle this information, and the more of it there is, the longer it takes.

Something else shows up immediately when you're dealing with the 'more is better' concept - just how long it takes is also dependent on the organization of the display memory in the computer's hardware. Bit-planes, like the Amiga uses, are terrific for certain types of display management like moving areas, and changing them in rectangular blocks like windows. Bit-planes are absolutely horrible to deal with if you need to actually do computations that involve individual pixels, though. When you make a 68000 or a 68020 manipulate one three-bit pixel from a bit-plane, the processor must handle literally hundreds of bits over multiple instructions to get this done - and locating and isolating the pixel costs even more in time and memory.

If the pixel is located in a pixel-plane instead of a bit-plane, and the pixel-plane is organized such that for each processor sized word (8, 16 or 32 bits for the Amiga) the processor can fetch, store, or otherwise operate on a single pixel in a single operation, immense improvements in computation times are achieved.

These computations are the kind that are done when you paint in an image and want to tint an area: each pixel needs to be modified individually based on it's previous value. If you wish to smear an area, each pixel is operated upon based upon its neighbours' values. These and similar operations cost a lot of processor time. The only reason that they run as fast as they do within the Amiga's bit-plane architecture is because the 68xxx series of processors is very powerful. When you want to handle a lot of data at one time, though, you'll run out of power and start paying in perceptible time delays.

Some ways to address these issues...

It is possible to design a display system such that the pixel information is available in both forms - as bit-planes, *and* as individual, atomic pixels. I must stress that this is ideal - and, of course, costs the absolute most in engineering skill, hardware density, and consumer dollars.

Most pixel-plane organizations are simply areas of memory that the processor accesses in the normal manner to deal with a particular pixel. Storage locations are unique locations in the processor's normal address space, arranged as a linear array from one bound of the display to another. There is another way to arrange a display, one that in my opinion is far superior, and not particularly costly in terms of engineering and material.

This is a cartesian pixel-plane. Here, the location of a pixel is specified by sending two values to the display hardware - these two values represent the X and Y position of the pixel on the display. This requires three operations by the processor: Send X, Send Y, and Operate (store, load, clear, increment, etc). An architecture like this can be expanded in such a way that the first two operations are eliminated more than 99 per cent of the time. In that case, we have even greater speed of access than a conventional pixel-plane array, while providing the processor cartesian access to the pixels - something most programmers would beg for if they considered it even lightly.

Having established that 24 bits to represent a single pixel is probably enough, and that it is convenient for a processor that can handle a 32-bit data item in a natural and efficient manner, let me go on to show you what you could do with the eight bits that are left over in that 32-bit word.

Let us imagine a display situation where the memory can be accessed in a cartesian pixel-plane. Two memory locations are used to specify an X and a Y position. One is used to specify the colour of the pixel, and one might be used for miscellaneous display operation commands (like a global erase in the current colour, set write mode to XOR, etc). A simple operation consists of storing X, then Y, then the pixel. This is reasonably efficient, certainly far more so than accessing a 24-bit pixel by handling 24 individual bits in 24 individual memory locations after creating the 24 different memory addresses. It may be more efficient than just loading the pixel from a normal memory location as well, if the address of the pixel needs to be calculated. One multiply instruction (common in pixel address calculations) would take longer than the entire three simple operations for the cartesian pixel-plane.

Suppose you want to write an area, though - this is very common, in fact a majority of display operations do this. Here's one situation where we can put those extra bits in the 32-bit word to extremely good use. We'll take four of the extra bits, and define them as *direction* bits.

If bit 28 is on, then after you write the pixel the display increments the X position for you. If bit 27 is on instead, then the display will decrement the X position for you. If bit 26 is on, the display will increment the Y position for you, and if bit 25 is on, the display will decrement the Y position for you.

You can specify combinations of these bits as well, resulting in x/y axis movement in any of eight directions. This is good, because it means that you can write a block of information anywhere on the screen, of any shape, by first storing one pair of x/y co-ordinates, and then writing a stream (using DMA, even!) of pixels to the pixel port which contain these inherent direction indicators. No address calculations, no *nothing* but move data to one address - which happens to be the single fastest thing the 68xxx series of processors can do when handling memory data. Put simply, the shape of the object is contained within the pixels themselves; it remains only for the processor to decide where to place the object (by writing the initial x/y address) and then slam the data in the memory as fast as possible.

The object creates itself. No matter how big, or what shape - think about it. There are four bits left, and one of them would become immediately useful as a 'no-write' bit; store a pixel with this set, and the x/y counters move, but no change is made to the display data. Three more bits left, and counting. Genlock! Two bits. You might use the other two bits as priority controls - that gives you four levels of priority for raster, sprite, whatever. These bits could be implemented in the pixels (at a cost of more memory) or just be current status indicators like extra command bits. I'm sure others could expand on this even further, and I hope they do.

But wait - it gets better! Highly efficient line, circle, ellipse, and other computation routines have been developed for use in an environment where the display position can be adjusted in any of eight equally distributed directions, like plotters - and this imaginary display. It turns out the these procedures involve integer addition and little else... something the 68xxx series, like most micro architectures, does quite well. Even a RISC architecture can still add!

An imaginary display? Well, not completely... When working in the video game industry, I designed a display that worked similarly to the one described here, except it had only eight colours available. It did implement the cartesian plane as discussed, and the direction control as well. I have yet to come across a display organization with the elegance and convenience of this configuration for processor manipulation of the data contained in the display. It is unfortunately becoming popular to consider the processor's access to the display data of secondary importance.

Why we need to look at this *now*

The IBM computers, with their terribly inefficient (and inappropriate for image processing/handling) segmented memory architecture, still have the vast majority of the market sewn up. The Mac and its kindred are wandering along behind the IBM and its clones, snapping up most of the individuals left in the dinosaur's wake with sharp, pointy marketing teeth.

The Amiga system is left to garner hard-core tech types who hid in the cracks while the parade went by, sneering and occasionally biting an ankle or two. Colourful, but not far off. Visionaries of several kinds are also attracted to the Amiga system, because they can see through the marketing glitz to some degree. I include game players and artists in this group.

The hardware to create such a display is available now; it's costly, but it is there. The main impediment to such a creation is the cost of memory in an architecture that is fast enough to handle video access simultaneously with processor/DMA access, while still forming an array dense enough to give us some reasonable kind of pixel density. The 'new' release of the Amiga's OS will be a major rewrite, and to an extent where major changes, such as the total number of bits it can handle (currently eight - ptooey) may be able to be designed in if the push is there to do it.

How much resolution? That's a toughie - at least 640, I'd rather see 1024 or even 2048 if it could be managed. For video (I hear you cry) you need far less horizontal resolution... except that television is an old, poor standard. Ask any video engineer and listen to them cry on your shoulder about the NTSC specification.

What about HDTV? Will the Amiga, with 320 horizontal resolution, be able to meet the challenge of the new modes? Also, is HDTV the end of things, or will we see even higher resolutions (I think we will as soon as the NTSC spec is buried - as it should have been years ago, in my opinion).

It does boil down to cost right now, for the resolution decision. But here's an interesting point - if the use of 16-bit variables were adopted for x/y values, then the OS would be unlikely to run out of dynamic range before the processor ran out of steam to fill an area that large (65536 x 65536) with that many pixels, and it costs very little extra memory compared to the current situation. It also allows large virtual windows. 32 bits might even be worthwhile. The thing is, if the decision is taken to limit the system to 16 bits for colour resolution, I believe that we'll all regret it very soon thereafter - just as developers are running up against the 8-bit limit of the current OS every day - and have been since the system was released.

The implementation of a display system with three distinct access modes - cartesian, linear, and bit-plane - would provide the computational and manipulative bandwidth that are lacking in any one of the individual systems.

If I were forced to choose one, I would immediately pick the cartesian plane and slave a processor to it, if I couldn't have bit-planes and a blitter too.

With virtual windows implemented, the display resolution could move up as the hardware became more practical. With 24-bit colour resolution handled, a vast silence would descend as artists, engineers, and scientists ceased complaining and set about using the extensive new ranges available - ranges which, no matter what else you might say about them, are likely to be... enough.

I hope this has got you thinking about these matters. Developers are quietly discussing these issues among themselves, and some of them are certainly thinking about it. If you have a particular viewpoint, be sure and let Those In Power know.

The Disk Mechanic

Give your disks a Tune-up

Only $89.95

The Disk Mechanic is a comprehensive collection of utilities for Amiga DOS, including TuneUp, DoubleBack, DiskRepair, and WorkShop.

TuneUp: With TuneUp you can improve hard disk or floppy disk performance by up to 400%. Not a caching program, TuneUp is a sophisticated disk optimizer that actually re-organizes the information on your disk to improve system efficiency. TuneUp is fully compatible with Amiga DOS.

DoubleBack: DoubleBack is a high speed archival hard disk back-up program. DoubleBack supports multiple source and destination drives, file selection by name, date, protection, directory, and file note. Back up a full 20 megabytes in under 40 minutes. Selectively copy files between hard disks and partitions. Protect your data!

Sample TuneUp benchmark of some standard 3.5" disks:

Volume Name	TuneUp	BEFORE	AFTER	IMPROVED
Workbench 1.3	3:30	4.88	2.70	79%
Digi!	3:45	9.44	5.22	80%
WordPerfect	4:42	12.36	5.37	130%
SuperBase	3:37	19.39	9.66	100%
data	1:43	14.30	4.68	205%
archives	3:40	28.43	9.52	299%

* Times given are the average of three runs on a standard A2000 internal floppy drive. The TuneUp time is the time required to process the disk (in minutes and seconds). The BEFORE and AFTER times are the time (in seconds) required to get a directory of the disk. The more use a disk has had, the better the improvement!

Demo disk Available; call for details.

DiskRepair: With DiskRepair it is possible to recover accidently deleted or discarded files, salvage files from damaged disks, and reconstruct a corrupted hard disk. DiskRepair works with hard and floppy disks, and is compatible with the Fast Filing System.

WorkShop: For the expert user, the disk WorkShop is a full featured disk file and block editor. Examine and modify disk files, search for viruses, and change menus or requesters in popular programs.

The Disk Mechanic also includes a dozen small CLI based utilities to make managing your disks easier. We are now shipping version 2.5 with full support for Amiga DOS 1.3 and the Fast Filing System. The Disk Mechanic is not copy protected.

Dealer inquiries welcome.
Amiga is a trademark of Commodore Amiga Inc.
WordPerfect is a trademark of WordPerfect Corp.
Digi! is a trademark of Aegis Development
SuperBase is a trademark of Precision Software, Ltd.

Lake Forest Logic Inc.
28101 E Ballard Road
Lake Forest, IL 60045
(312) 816-6666

MasterCard VISA

INTERCORE DEVELOPMENT SYSTEMS
7 DOVER HILL DRIVE
NESCONSET, NEW YORK 11767
(516) 361-8628

InterCore Development Systems Announces the Future of Telecommunications...

IDS MultiBoard

- o True MultiUser Operation
- o Fully Intuition Controlled
- o High Speed Dynamic Code
- o CB Simulator With 32 Channels
- o Real Time User Paging System
- o Baud Rates Up To 56,700 BPS
- o Automatic System Maintence
- o No Slow Down With Multiple Users
- o Create System Menus Easily With MenuMaker
- o Fully Multi-Tasking; Doesn't Hog CPU
- o Up To 32 Simultaneous Online Users
- o Console Screen Allows Easy Supervision
- o Advanced Message Editor with Word Wrap
- o View Individual Users; See Exactly What They Do
- o 65,535 Security Acess Levels; Individual RWUD for Each User
- o High Speed File Formats for a Suprisingly FAST System
- o XModem, XModem CRC, YModem, and YModem-Batch
- o 20 File Libraries & 20 Message Bases & Personal E-Mail System
- o Run A Single User System Through The Amiga's Standard Serial Port

We at Intercore Development Systems believe that we have brought the future of telecommunications into the home and workplace. IDS MultiBoard brings to the user power previously found only on expensive communication systems.

Whether for home or business use, IDS' MultiBoard contains features that will transform your Computer into a Communications Workstation. With abilities such as a Real-Time online CB Simulator you can hold online business conferences or just have a friendly conversation with a group of friends.

Although the IDS Multiboard is extremely powerful, it's also extremely easy to use. With everything accessible through pull-down menus and standard Intuition requestors, first time Users will be up and running in no time. Furthermore with the Automated Sysop Feature the system can run itself for weeks at a time.

Over two years in research and development the IDS MultiBoard is not only the best Communications Software for the Amiga, but for any Computer System period! So ask your local Authorized Amiga Dealer to show you a demonstration or call Intercore Development Systems at (516) 361-8628 today.

For Multi-User Operation, use Amigo Business Computer's ComPort.

Dealers Inquiries Welcome.

Amiga is a Trademark of Commodore-Amiga, Inc. ComPorts is a Trademark of Amigo Business Computers.

Getting A Handle On File Systems

Part 2 of a series

by John Toebes and Doug Walker
Copyright © 1989 Doug Walker and John A. Toebes, VIII.

Doug Walker is a Senior Systems Developer at SAS Institute, Inc. in Cary, North Carolina. He has been programming the Amiga since 1986 as a member of The Software Distillery, which is responsible for many popular freely redistributable programs including Hack and Hack Lite, PopCLI, MemWatch and Blink. He can be reached on PeopleLink (DWALKER) and Usenet (mcnc!rti!sas!walker).

John Toebes is Manager of the Motorola Compiler Department at SAS Institute, Inc., responsible for the recent release of the Lattice C compiler. He is also coordinator for the Software Distillery. He can be reached on Bix (JTOEBES) and Plink (JTOEBES) as well as Usenet (mcnc!rti!sas!toebes).

Last time we discussed handlers and file systems from the application program's point of view. This article will cover writing your own file system, with examples taken from a Software Distillery implementation of a file system to read and write AmigaDOS 1.2 format floppies. The full source code to this implementation is available on the Transactor disk. (Note: Since a file system is just a special case of a handler, we will use the terms interchangeably.)

How DOS knows about your file system

With the exception of a few pre-known handlers, a file system is made known to AmigaDOS through an entry in the global device list. This list, which is maintained in the DosInfo structure located through DosLibrary, contains entries for all devices, volumes and assigns available to the user.

Because not all handlers may be needed at any given time, AmigaDOS supports the concept of loading them on demand. It is able to determine the status of any handler by looking at its corresponding DeviceNode entry in the device list. This structure (shown in Figure 1) contains everything that AmigaDOS needs to locate the handler if it is running and to start it up when it is not.

DeviceNodes are placed on the device list in one of two ways:

1) Through a call to expansion.library (generally done by the ROM code on an autoconfiguring board).

2) Through a user invocation of the Mount command. We will not concern ourselves with this first case as it is not generally useful for non-hardware specific handlers.

```
struct DeviceNode {
    BPTR    dn_Next;        /* singly linked list                        */
    ULONG   dn_Type;        /* always 0 for dos "devices"                */
    struct MsgPort *dn_Task; /* standard dos "task" field.  If this is
                             * null when the node is accesses, a task
                             * will be started up                        */
    BPTR    dn_Lock;        /* not used for devices -- leave null        */
    BSTR    dn_Handler;     /* filename to loadseg (if seglist is null) */
                            /* HANDLER=   FILESYSTEM=                    */
    ULONG   dn_StackSize;   /* stacksize to use when starting task       */
                            /* STACKSIZE=                                */
    LONG    dn_Priority;    /* task priority when starting task          */
                            /* PRIORITY=                                 */
    BPTR    dn_Startup;     /* startup msg: FileSysStartupMsg for disks */
    BPTR    dn_SegList;     /* code to run to start new task (if necessary)
                             * if null then dn_Handler will be loaded.  */
    BPTR    dn_GlobalVec;   /* set to -1 to indicate non-BCPL program    */
    BSTR    dn_Name;        /* the node name, e.g. '\3','D','F','3'      */
                            /* Device name that was mounted              */
};
```

Figure 1

When the Mount command processes a device, it allocates a DeviceNode structure and fills in the corresponding fields in the structure for each keyword in the mountlist. In the special case of a file system (identified by the 'filesystem=' option), Mount will allocate a FileSystemStartupMsg (Figure 2) in order to store the additional configuration information. (Do not let the name confuse you, as no messages are involved with this structure)

```
struct FileSysStartupMsg {
    ULONG   fssm_Unit;      /* exec unit number for this device          */
                            /* UNIT=                                     */
    BSTR    fssm_Device;    /* null terminated bstring to the device name */
                            /* DEVICE=                                   */
    BPTR    fssm_Environ;   /* ptr to environment table (see above)      */
    ULONG   fssm_Flags;     /* flags for OpenDevice()                    */
                            /* FLAGS=                                    */
};
```

Figure 2

Once the Mount command has done its work, a handler is not pulled into action until it is referenced. Within the DOS Library, all calls that deal with a file name are routed through the routine DeviceProc(), which is responsible for matching up a

handler with the given name. It does this by searching the device list until it encounters an entry with the corresponding name. When confronted with an entry that it determines to be a device, it checks the dn_Task field to see if the handler is currently running.

If the dn_Task field is null, DeviceProc() calls LoadSeg() to bring the code for the handler using the name given in the dn_Handler field. If successful, it stores a pointer to the seglist into the dn_Seglist field and then creates a new process with the stack size and priority stored in the device note. (Note that we will assume that the dn_GlobalVec field is set to -1 indicating that the program does not use BCPL conventions.)

File system initialization

Once the process has been successfully started, DeviceProc() will send it a standard packet with dp_Arg1 pointing to the device name and dp_Arg3 pointing to the device node. The packet type will be uninitialized although it generally contains the priority that the handler was started with. Because this is not a true DOS packet, it must be processed separately by the handler.

During initialization, a file system must fill in the dn_Task field of the DeviceNode structure so that all subsequent requests for the device are sent to the same task. Handlers such as CON: should not fill in the dn_Task field because each open should resolve to a separate window and hence a separate handler. If for any reason during initialization the handler is unable to properly complete initialization, it should return DOSFALSE in dp_Res1 and then die quietly. Only upon successful initialization should it return DOSTRUE.

As a part of the initialization, the file system should open all devices and allocate any memory necessary for housekeeping and buffers. A file system should obtain the configuration information from the DosEnvec array (Figure 3) located off the File System Startup Message (found by looking in the dn_Startup field of the DeviceNode structure). It is important to note that this is not an ordinary array. To allow for future compatibility and some level of defaulting, only enough entries to hold the information specified in the mount list is allocated. The number of entries actully present in the array is indicated by the first field, requiring you to check before accessing an element. For example, if you wanted to look at the BUFMEMTYPE entry (number 12) you would need to ensure that the table size is at least 12. In order to eliminate this testing in our code, we simply allocate a full-sized array initialized with our defaults and then copy over those entries that were presented to us.

Because this configuration information is supplied by the user in the form of a text file read by the Mount command, it is important that the initialization code enforce a level of sanity to protect itself from any mistakes. This sanity checking should also be done throughout the handler due to the extra level assumed for any system functionality.

```
struct DosEnvec {
    ULONG de_TableSize;      /* Size of Environment vector                 */
    ULONG de_SizeBlock;      /* in longwords: standard value is 128        */
                             /* In MountList: BLOCKSIZE=                   */
    ULONG de_SecOrg;         /* not used; must be 0                        */
    ULONG de_Surfaces;       /* # of heads (surfaces). drive specific      */
                             /* SURFACES=                                  */
    ULONG de_SectorPerBlock; /* not used; must be 1                        */
    ULONG de_BlocksPerTrack; /* blocks per track. drive specific           */
                             /* BLOCKSPERTRACK=                            */
    ULONG de_Reserved;       /* DOS reserved blocks at start of partition */
                             /* RESERVED=                                  */
    ULONG de_PreAlloc;       /* DOS reserved blocks at end of partition   */
                             /* PREALLOC=                                  */
    ULONG de_Interleave;     /* usually 0                                  */
                             /* INTERLEAVE=                                */
    ULONG de_LowCyl;         /* starting cylinder. typically 0             */
                             /* LOWCYL=                                    */
    ULONG de_HighCyl;        /* max cylinder. drive specific               */
                             /* HIGHCYL=                                   */
    ULONG de_NumBuffers;     /* Initial # DOS of buffers.                  */
                             /* BUFFERS=                                   */
    ULONG de_BufMemType;     /* type of mem to allocate for buffers        */
                             /* BUFMEMTYPE=                                */
    ULONG de_MaxTransfer;    /* Max number of bytes to transfer at a time */
                             /* MAXTRANSFER=                               */
    ULONG de_Mask;           /* Address Mask to block out certain memory   */
                             /* MASK=                                      */
    LONG  de_BootPri;        /* Boot priority for autoboot                 */
                             /* BOOTPRI=                                   */
    ULONG de_DosType;        /* ASCII (HEX) string showing filesystem type;
                              * 0X444F5300 is old filesystem,
                              * 0X444F5301 is fast file system             */
                             /* DOSTYPE=                                   */
};
```

Figure 3

It is not necessary for a file system to be written re-entrantly because it only acts as a server processing messages and is never called directly from an outside source. Handlers such as CON: must be re-entrant because the seglist stored in the DeviceNode is shared amongst multiple processes. It is necessary for handlers and file systems to not assume that they were started like any normal CLI program because A0 and D0 will not contain anything even resembling a command line.

Debugging file systems

The environment that the handler starts up with also brings an interesting twist into what can be done from a handler. Because a handler recieves all its requests through the message port in the process structure, it is unable to utilize the standard AmigaDOS routines Open(), Read(), etc., which use the same port. In fact, if you attempt to do so, the odds are very likely that you will see Guru 87000004. In addition, because the handler is started asynchronously, it becomes extremely difficult to debug. In particular, it is the initialization process which is the most problematic to debug. For serious problems in initialization, it is useful to use requesters to temporarily halt processing and provide a diagnostic of the current situation. With our implementation, we have engineered the technology to write debugging strings to a window, which greatly aids in debugging the handler. We hope this method will be adopted by all future handler-writers as a convenient, useful standard.

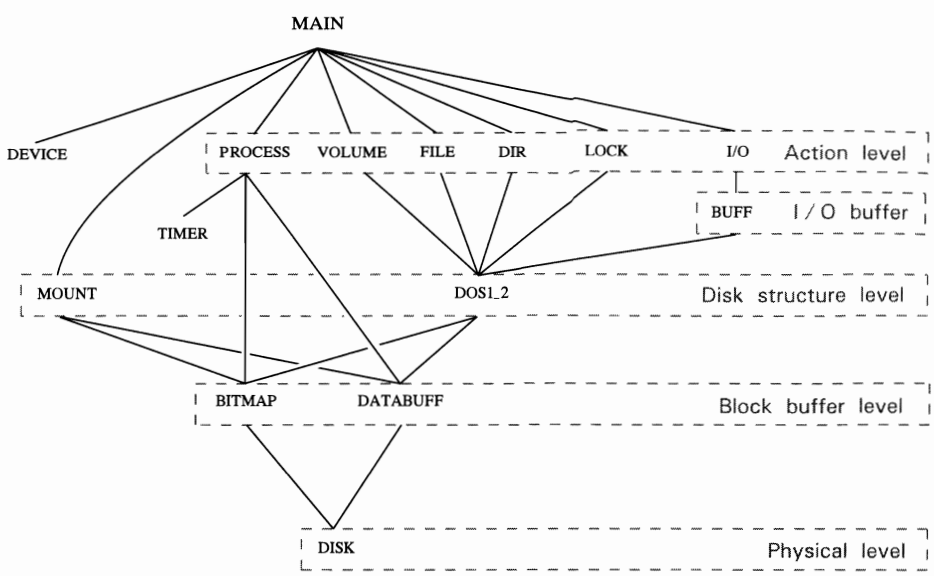


Figure 4: *organization of the Distillery's 1.2 floppy file system*

The ACTION_HANDLER_DEBUG packet defines the debugging environment. The dp_Arg1 field contains a debugging level. If it is nonzero, dp_Arg2 should contain a BPTR to a file handle as obtained from Open(). Normally this file handle is the result of using Open() on a CON: window specification. The handler can safely write debugging output to this file via the DOS packet interface. If dp_Arg1 is zero, debugging is to be cancelled. No value is passed in dp_Arg2. In either case, the handler should return DOSTRUE in dp_Res1 and the old debugging file handle (or NULL) in dp_Res2. This allows the external program to close the old filehandle.

Notice that the only reserved debugging level is 0, meaning no debugging. The handler is free to define the meaning of nonzero values as it wishes, with the understanding that it must keep track of the file handle provided in dp_Arg2 so it can return it in a later ACTION_HANDLER_DEBUG. If a debugging file is open when the handler terminates, it should close it using the ACTION_TERMINATE packet.

Volumes and locks

AmigaDos has the concept of a 'volume' of storage associated with a device. A volume can be one partition of a hard disk, one floppy disk, or whatever else the file system decides to make it. Since our handler works with floppy disks, we will need to be able to tell DOS about new volumes as they are inserted, and let DOS know when they are no longer available.

When a disk is inserted, we add a struct DeviceList to the device list discussed earlier corresponding to the new volume. The dl_Task field of the DeviceList structure points back to the file system so DOS will know where to send requests dealing with the volume. As the volume is used, applications will create locks on files on the volume, indicating that they are in use.

If no locks are outstanding when the volume is ejected, the file system deletes the volume node from the list. If there are outstanding locks, the file system collects them into a linked list and attaches the list to the volume node. It then sets the dl_Task field to NULL, indicating that the volume is not owned by any file system (and is therefore unavailable).

If the volume is re-inserted later, the file system checks the volume list for an entry corresponding to the volume. If it finds one, it sets the dl_Task field to point to itself and takes over the list of locks.

The functions Mount() and Dismount() in the file Volume.c maintain the volume list in our handler.

As mentioned earlier, locks are created on files to indicate they are in use. For each lock created, a struct FileLock is allocated and added to the linked list of all outstanding locks on the volume. Struct FileLock is defined in <libraries/dosextens.h> and looks like this:

```
struct FileLock {
    BPTR            fl_Link;        /* bcpl pointer to next lock */
    LONG            fl_Key;         /* disk block number         */
    LONG            fl_Access;      /* exclusive or shared       */
    struct MsgPort  fl_Task;        /* handler task's port       */
    BPTR            fl_Volume;      /* bptr to a DeviceList      */
};
```

The fl_Link field is a BCPL pointer pointing to the next lock in the chain. The fl_Key field is handler dependant. In the 1.2 handler, this field contains a disk block number associated with the file being locked. This is not a requirement, however, and handlers may put any value they like in this field that allows them to uniquely identify the file being locked. The fl_Access field is either ACCESS_EXCLUSIVE or ACCESS_SHARED, indicating whether the lock is for write or read, respectively. The fl_Task field points to the file system associated with the lock, and the fl_Volume field points to the associated volume node.

Implementation overview

Now that we have an idea of what the file system is going to do, let's take a more detailed look at the Distillery's 1.2 floppy file system.

The file system is organized into several layers, as depicted in Figure 4. The top layer is contained totally in the file main.c. This file should be totally portable to any file system you care to write. We have used the same main.c for the 1.2 floppy file

system, an MS-DOS file system and a network file system. This layer calls into the second layer of routines, which are distributed across the files Device.c, Process.c, Volume.c, File.c, Dir.c, Lock.c and IO.c. This layer contains one function for each of the AmigaDOS packet types described in the previous article. The entire second layer should be portable to most, if not all, file systems. The disk structure layer, Mount.c and Dos1_2.c, is the first handler-dependent layer. It contains routines to manipulate the layout of the file information on top of the blocks of the disk. These routines provide generic operations such as locate file, examine information, change information or access file contents.

The disk structure layer routines access information through a virtual buffering scheme managed by the block buffer level. All routines assume that only one buffer is available at any time. This level is responsible for caching and flushing blocks as necessary. The physical layer contains the only routines that actually access the physical media.

By layering the code in this manner, we can easily modify major pieces without affecting other code. With one small exception, the Action, I/O buffer and Block Buffer layers are identical for the MS-DOS and AmigaDOS handlers.

Getting to the handler

In looking at a handler, it is helpful to have a understanding of how a normal C program call to manipulate a file is translated and transmitted to the appropriate handler. To illustrate, let us take a simple statement:

```
file = fopen("vol:dir/name.ext", "r");
```

Using a typical UNIX implementation, fopen() would allocate some data structures and then turn around and call open() to access the file; open(), in turn, does some housekeeping and eventually gets down to calling the AmigaDOS routine found in DOS.LIBRARY:

```
Open("vol:dis/name.exe", MODE_OLDFILE);
```

Once in this routine, AmigaDOS must perform several tasks:

1) Call DeviceProc() with the given name in order to locate the handler responsible. DeviceProc() brings any unloaded handlers into memory, as well as associating volume or device node entries with the appropriate handler. In the case of an unknown volume name, DeviceProc() will put up a requester asking the user to insert a disk in any drive. If after this searching it is unable to locate a handler for the name, it will return NULL and Open() will terminate with a failure code.

2) Given a handler, Open() will then allocate memory for the FileHandle and fill in the default fields in the handle. In the event of no memory, Open() will return the failure back to the application.

3) Using a routine akin to the sendpkt() routine in the last issue, Open() will then send the packet to the handler with the Arg fields filled in appropriately. For the action number on the packet, Open() will use the mode passed by the application. In filling in the arguments, it must convert the incoming C string to an equivalent BSTR by copying the string to a separate buffer and prepending it with a length byte.

On the handler end, all it sees is the incoming ACTION_OPENINPUT with the appropriate arguments. It is the responsibility of the handler to parse the name completely, as it will come exactly as specified by the user. The handler must be able to handle names that traverse directories (as in our case) as well as the special meaning of a leading '/' as going up a directory. It must also check for any restrictions on opening the file, such as locks or protection attributes.

4) Once the packet has returned, Open() examines the result. In the event of a failure return from the handler, Open() deallocates the file handle and returns a NULL to the application. Otherwise, it will return a BPTR to the file handle that it allocated.

Given this returned file handle, the remaining routines in the DOS Library can locate the handler by simply looking in the fh_Type field for the corresponding handler. This is important because once a handler has initiated an open of a file, no other handler can perform the operations on that file. If a request is made to operate on a file for a volume which the handler does not have mounted, it will return a failure code to DOS, which will put up a requester demanding the disk in the corresponding drive.

Once Close() is finally called, DOS will free the memory allocated by Open().

Note that there is no requirement for an application to use the Open() routine in the DOS Library; it is quite possible to write a compatible one in C. However, the DOS Library routines are quite robust, understanding volume full and I/O errors, allowing it to attempt to retry the operation when possible.

Top layer: main.c

Main.c (see Listing 1) has four primary functions. It calls the necessary initialization and termination routines; It receives and replies to application packet requests; it does some table-driven translation on the input arguments to get rid of some of the annoying BCPL problems that are inevitable when writing a file system in C; and it decides which of the second-layer routines to call to process the packets as they come in.

The BCPL translation is an interesting trick that requires further discussion. Recall that many of the incoming arguments to the packets are BCPL pointers. Rather than have all the second-level routines convert the BCPL pointers to C pointers before using them, we chose to set up a table of packet types. One field in the table indicates which of the four arguments should be translated, if any. If the table indicates that an argument should be translated, main.c converts it to a BCPL string before passing it on to the second-level routine.

The same table also contains a pointer to the second-layer function that processes the packet. Thus, with a single table lookup, we determine what arguments to translate and what function to call when we are done. The only drawback to this method is that the packet type numbers are not contiguous. They do come in blocks, however, with the numbers 0-34 and 1004-1008 defined, and some random exceptions to all the rules. Main.c therefore contains a switch statement to choose which array to look the table entry up in.

The sequence of events in main.c is then:

1. Initialize, including mounting a volume node for the current volume if there is one.

2. Loop while processing packets from applications:
   A. Wait for a packet.
   B. When a packet comes in, perform BCPL translation and call the appropriate second-layer function.
   C. Return the packet to the application's ReplyPort 3. When the ACTION_DIE packet is received, call the termination routines.

Since the handler does check its MsgPort nearly continuously, the MsgPort is a convenient place to send additional handler information. Three packets in particular may be used to allow the handler to perform maintenance. Applications should not send these packet types.

ACTION_TIMER_PACKET: The handler normally sets up timer request packets that return every few seconds. This wakes up the handler and allows it to perform time-critical maintenance such as flushing disk buffers.

ACTION_READ_RETURN and ACTION_WRITE_RETURN: If the handler is doing asynchronous I/O, the lower-level I/O device will return one of these packets on completion of the I/O. When the handler receives the ACTION_READ_RETURN or ACTION_WRITE RETURN packet, it knows it is safe to return the original packet to the application requesting the read or write.

Since all three of these packet types are really replies to messages generated by the handler, the handler should not reply to them as it does to application messages.

Action layer: lock.c

Let's look at one of the more interesting layer 2 files as an example. Lock.c (see Listing 2) contains layer 2 routines to handle the ACTION_LOCATE_OBJECT (Lock), ACTION_COPY_DIR (DupLock) and ACTION_FREE_LOCK (UnLock) packet types. The file should be usable as is for any handler. The key (pun intended) to this handler independence is the fact that there exists by definition a one-to-one correspondence between files and keys. This allows a routine that knows nothing of the meaning of the key to determine if a file is already locked simply by examining the keys of all outstanding locks.

A utility routine, CreateLock(), is defined to handle most of the ugliness associated with lock manipulation. When any of the Action routines needs to come up with a lock structure, they call CreateLock(), passing it the desired mode and the key of the file to lock. CreateLock() examines all outstanding locks on the volume and determines if the lock request is legal; if so, it allocates and initializes a struct FileLock, and if not, it returns an error.

The presence of CreateLock() makes the functions ActLock() and ActDupLock() ridiculously simple. The code for these functions is listed in Listing 2. ActLock() calls layer 3 to get the key associated with the file, then calls CreateLock(). ActDupLock() gets the key from the old lock, then calls CreateLock() on that key.

Other layer 2 files are similarly designed, with calls into the I/O buffer or Disk Structure layers to perform handler-dependent functions.

Windup

Hopefully, we have provided you with enough information to get you started on your own handler. If you need further assistance, feel free to contact the Software Distillery.

Next month we will trace the layering further, showing the I/O layers for both AmigaDOS and MS-DOS diskettes.

* * *

Acknowledgement

The Software Distillery file system is a collective effort of the Distillery, with major contributions by John and Doug, John Mainwaring, Jim Cooper, Dave Baker, Bruce Drake and Gordon Keener. Special thanks are due to Steve Beats and Andy Finkel of Commodore Business Machines for putting up with our nitnoid questions.

Listing 1: *main.c*

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* |_o_o|\\ Copyright (c) 1988 The Software Distillery.  All Rights Reserved */
/* |. o.| || This program may not be distributed without the permission of   */
/* | .  | || the authors:                                BBS: (919) 481-6436   */
/* | o  | ||   John Toebes     John Mainwaring    Jim Cooper                 */
/* |  . |//   Bruce Drake     Gordon Keener      Dave Baker                 */
/* ======                                                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

#include "handler.h"

/*****************************************************************************/
/******************** Dispatch table to handle all packets *******************/
/*****************************************************************************/
#define BP1 1
#define BP2 2
#define BP3 4
#define BP4 8

typedef void (*ifuncp)(GLOBAL, struct DosPacket *);
struct LookupTable
   {
   ifuncp subr;
   int flags;
   };

#define LO_FIRST 0
#define LO_LAST  34
struct LookupTable lowork[LO_LAST+1] = {
   { NULL,             0  | 0  | 0  | 0   }, /*  0 - ACTION_NIL           */
   { NULL,             0  | 0  | 0  | 0   }, /*  1 - Unknown              */
   { NULL,             BP1| BP2| BP3| 0   }, /*  2 - ACTION_GET_BLOCK     */
   { NULL,             0  | BP2| BP3| 0   }, /*  3 - Unknown              */
   { NULL,             BP1| BP2| BP3| 0   }, /*  4 - ACTION_SET_MAP       */
   { ActDie,           0  | 0  | 0  | 0   }, /*  5 - ACTION_DIE           */
   { NULL,             0  | 0  | 0  | 0   }, /*  6 - ACTION_EVENT         */
   { ActCurentVol,     BP1| 0  | 0  | 0   }, /*  7 - ACTION_CURRENT_VOLUME */
   { ActLock,          BP1| BP2| 0  | 0   }, /*  8 - ACTION_LOCATE_OBJECT  */
   { ActRenameDisk,    BP1| BP2| 0  | 0   }, /*  9 - ACTION_RENAME_DISK    */
   { NULL,             0  | 0  | 0  | 0   }, /* 10 - Unknown              */
   { NULL,             0  | 0  | 0  | 0   }, /* 11 - Unknown              */
   { NULL,             0  | 0  | 0  | 0   }, /* 12 - Unknown              */
   { NULL,             0  | 0  | 0  | 0   }, /* 13 - Unknown              */
   { NULL,             0  | 0  | 0  | 0   }, /* 14 - Unknown              */
   { ActUnLock,        BP1| 0  | 0  | 0   }, /* 15 - ACTION_FREE_LOCK     */
   { ActDelete,        BP1| BP2| 0  | 0   }, /* 16 - ACTION_DELETE_OBJECT  */
   { ActRename,        BP1| BP2| BP3| BP4 }, /* 17 - ACTION_RENAME_OBJECT  */
   { NULL,             0  | 0  | 0  | 0   }, /* 18 - ACTION_MORE_CACHE    */
   { ActDupLock,       BP1| 0  | 0  | 0   }, /* 19 - ACTION_COPY_DIR      */
   { NULL,             0  | 0  | 0  | 0   }, /* 20 - ACTION_WAIT_CHAR     */
   { ActSetProtection, 0  | BP2| BP3| 0   }, /* 21 - ACTION_SET_PROTECT   */
   { ActCreateDir,     BP1| BP2| 0  | 0   }, /* 22 - ACTION_CREATE_DIR    */
   { ActExamine,       BP1| BP2| 0  | 0   }, /* 23 - ACTION_EXAMINE_OBJECT */
   { ActExNext,        BP1| BP2| 0  | 0   }, /* 24 - ACTION_EXAMINE_NEXT  */
   { ActDiskInfo,      BP1| 0  | 0  | 0   }, /* 25 - ACTION_DISK_INFO     */
   { ActInfo,          BP1| BP2| 0  | 0   }, /* 26 - ACTION_INFO          */
   { ActFlush,         0  | 0  | 0  | 0   }, /* 27 - ACTION_FLUSH         */
   { ActSetComment,    0  | BP2| BP3| BP4 }, /* 28 - ACTION_SET_COMMENT   */
   { ActParent,        BP1| 0  | 0  | 0   }, /* 29 - ACTION_PARENT        */
   { ActTimer,         BP1| 0  | 0  | 0   }, /* 30 - ACTION_TIMER         */
   { ActInhibit,       0  | 0  | 0  | 0   }, /* 31 - ACTION_INHIBIT       */
   { NULL,             BP1| 0  | 0  | 0   }, /* 32 - ACTION_DISK_TYPE     */
   { NULL,             0  | 0  | 0  | 0   }, /* 33 - ACTION_DISK_CHANGE   */
   { NULL,             0  | 0  | 0  | 0   }  /* 34 - ACTION_SET_FILE_DATE  */
     };

#define HI_FIRST 1004
#define HI_LAST  1008
struct LookupTable hiwork[5] = {
   { ActFindwrite,     BP1| BP2| BP3| 0   }, /* ACTION_FIND_WRITE  - 1004 */
   { ActFindin,        BP1| BP2| BP3| 0   }, /* ACTION_FIND_INPUT  - 1005 */
   { ActFindout,       BP1| BP2| BP3| 0   }, /* ACTION_FIND_OUTPUT - 1006 */
   { ActEnd,           0  | 0  | 0  | 0   }, /* ACTION_END         - 1007 */
   { ActSeek,          0  | 0  | 0  | 0   }  /* ACTION_SEEK        - 1008 */
   };

struct DosLibrary *DOSBase;

void _main()
{
   struct DosPacket   *mypkt;      /* a pointer to the dos packet sent       */
   int                action;
   ifuncp             subr;
   int                flags;
   struct global      global;

   DOSBase = (struct DosLibrary *)OpenLibrary(DOSNAME,0);

   /* Initialize our global data structure */
   memset((char *)&global, 0, sizeof(struct global));
   global.self   = (struct Process *) FindTask(0L);  /* find myself        */
   global.run    = 1;
   global.port   = &(global.self->pr_MsgPort);
                 /* install our taskid ...   */


   /* Initialize the intuitext structures for the requesters we might have   */
   /* to display                                                             */
   /* Because we have no scruples we can cheat and do this with a couple of  */
   /* long word assignments.  We leave the acual C code commented out here   */
   /* so that if this structure ever changed we will still be able to work   */
#if 0
   global.line1.FrontPen = global.line1.BackPen = -1;
   global.line1.DrawMode = JAM1;
   global.line1.LeftEdge = global.line1.TopEdge = 4;
   global.line2 = global.line1;
   global.line3 = global.line1;
   global.retrytxt = global.line1;
   global.canceltxt = global.line1;
#else
   *(long *)&global.line1.FrontPen     = 0x00010000L | (JAM1<<8);
   *(long *)&global.line1.LeftEdge     = 0x00040004L;  /* 4,4  */
   *(long *)&global.line2.FrontPen     = 0x00010000L | (JAM1<<8);
   *(long *)&global.line2.LeftEdge     = 0x0004000EL;  /* 4,14 */
   *(long *)&global.line3.FrontPen     = 0x00010000L | (JAM1<<8);
   *(long *)&global.line3.LeftEdge     = 0x00040018L;  /* 4,24 */
   *(long *)&global.retrytxt.FrontPen  = 0x00010000L | (JAM1<<8);
   *(long *)&global.retrytxt.LeftEdge  = 0x00040004L;
   *(long *)&global.canceltxt.FrontPen = 0x00010000L | (JAM1<<8);
   *(long *)&global.canceltxt.LeftEdge = 0x00040004L;
#endif
   global.retrytxt.IText = "Retry";
   global.canceltxt.IText = "Cancel";

   /* since we were started as a non-BCPL module we get sent the parameter */
   /* packet (ie. parameter packet not in D1) */

   mypkt = taskwait(&global);  /* wait for parameter packet */

#ifdef DEBUG
   initdebug();
#endif

   global.devname  = (((char *)BADDR(mypkt->dp_Arg1))+1);
                                        /* BSTR name passed to handler */
#if 0
   global.extra    = mypkt->dp_Arg2;      /* Extra Info passed          */
#endif

   /* get pointer to our device node */
   global.node = (struct DeviceNode *) BADDR(mypkt->dp_Arg3);
   global.node->dn_Task = global.port;

   InitDevice(&global);

   /* Now mount the disk */
   Mount(&global);

   OpenTimer(&global);

   mypkt->dp_Res1 = DOS_TRUE;
   retpkt(&global, mypkt);
```

```
PostTimerReq(&global);

while(global.run)   /* start of the real work */
   {
   mypkt = taskwait(&global);  /* wait for a packet */
   action = mypkt->dp_Type;

   BUG(("Execute: action #%ld\n", action));

   switch (action)
      {
      case ACTION_READ:
         subr = ActRead;
         flags = 0;
         break;
      case ACTION_WRITE:
         subr = ActWrite;
         flags = 0;
         break;

      case ACTION_SET_RAW_MODE:
         subr = NULL;
         flags = 0;
         break;

      case ACTION_FIND_WRITE:  /* 1004 */
      case ACTION_FIND_INPUT:  /* 1005 */
      case ACTION_FIND_OUTPUT: /* 1006 */
      case ACTION_END:         /* 1007 */
      case ACTION_SEEK:        /* 1008 */
         subr = hiwork[action-HI_FIRST].subr;
         flags = hiwork[action-HI_FIRST].flags;
         break;

      default:
         if ((action >= LO_FIRST) && (action <= LO_LAST))
            {
            subr = lowork[action-LO_FIRST].subr;
            flags = lowork[action-LO_FIRST].flags;
            }
         else
            subr = NULL;
      }

   mypkt->dp_Res1 = DOS_FALSE;
   mypkt->dp_Res2 = ERROR_ACTION_NOT_KNOWN;

   if (subr != NULL)
      {
      global.reply = 1;
      if (flags & BP1)
         mypkt->dp_Arg1 <<= 2;
      if (flags & BP2)
         mypkt->dp_Arg2 <<= 2;
      if (flags & BP3)
         mypkt->dp_Arg3 <<= 2;
      if (flags & BP4)
         mypkt->dp_Arg4 <<= 2;

      if (global.changedisk)
         Mount(&global);

      (*subr)(&global, mypkt);

      if (global.changedisk)
         Mount(&global);
      }
   else
      {
      BUG(("Unknown packet type %ld\n",mypkt->dp_Type));
      }

   /* Now return the packet to them */
   if (global.reply)
      retpkt(&global, mypkt);

   BUG(("-----\n"));
   }

   /* do our final cleanup */
   global.node->dn_Task = FALSE; /* zero the taskid field of device node */
   global.node->dn_SegList = 0;  /* make us be gone */

   DisMount(&global);
   TermDevice(&global);
   FreeBlkBuffs(&global);
}
```

Listing 2: *lock.c*

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/* |_o_o|\\ Copyright (c) 1987 The Software Distillery.  All Rights Reserved */
/* |. o.| || This program may not be distributed without the permission of   */
/* | .  | || the authors:                                          BBS:      */
/* | o  | ||   John Toebes     Dave Baker     John Mainwaring                  */
/* |  . |//                                                                  */
/* ======                                                                     */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */

/* Lock.c - lock manipulation */
/* ActLock, ActDupLock, ActUnLock */

/*-----------------------------------------------------------------------*/
/* Structure of a Lock:                                                  */
/*   struct FileLock {                                                   */
/*      BPTR fl_Link;             Next lock in the chain of device locks  */
/*      LONG fl_Key;              Block number of directory or file header */
/*      LONG fl_Access;           Shared Read (-2) or Exclusive Write (-1) */
/*      struct MsgPort * fl_Task; Handler process for Lock (Us)           */
/*      BPTR fl_Volume;           Node in DevInfo structure for Lock      */
/*      };                                                                */
/*-----------------------------------------------------------------------*/
#include "handler.h"

long GetKey(global,lock)
struct global *global;
struct FileLock *lock;
{
   if (global->volume == NULL)
      {
      global->pkt->dp_Res2 = ERROR_DEVICE_NOT_MOUNTED;
      return(0);
      }

   if (lock == NULL)
      return(global->Root);
   return(lock->fl_Key);
}

/* create a lock to be returned */
struct FileLock *CreateLock(global, key, mode)
GLOBAL global;
KEY key;
int mode;
{
struct FileLock *lock;
int access;

   lock = NULL;

   if (global->volume == NULL)
      {
      global->pkt->dp_Res2 = ERROR_DEVICE_NOT_MOUNTED;
      key = 0;
      }

   if (key)
      {
      /* if we are creating a write lock, make sure we can lock the disk */
      access = 0;

      BUG(("Searching for locks from %08lx\n", BADDR(global->volume->dl_Lock)));

      Forbid();
      /* Now run through the list of locks for this volume to figure out */
      /* What level of access it is currently locked for                 */
```

```
    for (lock = (struct FileLock *)BADDR(global->volume->dl_Lock);
         lock != NULL && access == 0;
         lock = (struct FileLock *)BADDR(lock->fl_Link))
       {
       /* Does it match what we want to access? */
       if (lock->fl_Key == key)
          access = lock->fl_Access;
       }
    Permit();

    BUG(("Access outstanding is %ld\n", access));

    /* Given the prevailing access mode, see if we will allow the new lock */
    switch(mode)
       {
       case SHARED_LOCK:
          if (access == EXCLUSIVE_LOCK)
             {
             /* Not legal, tell them so... */
             global->pkt->dp_Res2 = ERROR_OBJECT_IN_USE;
             return(NULL);
             }
          break;

       case EXCLUSIVE_LOCK:
          /* Does anyone else have it open? */
          if (access)
             {
             /* Not legal, tell them so... */
             global->pkt->dp_Res2 = ERROR_OBJECT_IN_USE;
             return(NULL);
             }
          break;

       default:
          BUG(("Invalid access %ld requested for key %ld\n", mode, key));
          global->pkt->dp_Res2 = ERROR_INVALID_LOCK;
          return(NULL);
       }

    if ((lock = (struct FileLock *)
                DosAllocMem(global, sizeof(struct FileLock))) != NULL)
       {
       BUG(("Creating the lock %08lx for key=%ld\n", lock, key));

       /* now fill in the lock */
       lock->fl_Key = key;
       lock->fl_Access = mode;
       lock->fl_Task = global->port;
       lock->fl_Volume = MKBADDR(global->volume);

       Forbid();
       lock->fl_Link = global->volume->dl_Lock;
       global->volume->dl_Lock = MKBADDR(lock);
       Permit();

       BUG(("Task=%08lx Vol=%08lx\n", lock->fl_Task, lock->fl_Volume));
       }
    }
  return(lock);
}

void freelock(global, lock)
GLOBAL global;
struct FileLock *lock;
{
struct FileLock *this;

  /* freeing NULL (root lock) is a bad move    */
  if (lock == NULL)
     {
     BUG(("NULL Lock\n"));
     return;
     }

  if (global->volume == NULL)
     {
     global->pkt->dp_Res2 = ERROR_DEVICE_NOT_MOUNTED;
     return;
     }

  /* locate the lock on the global volume list */
  Forbid();

  if ((this = (struct FileLock *)BADDR(global->volume->dl_Lock)) == lock)
     global->volume->dl_Lock = lock->fl_Link;
  else
     {
     while(this != NULL && (struct FileLock *)BADDR(this->fl_Link) != lock)
        this = (struct FileLock *)BADDR(this->fl_Link);
     if (this == NULL)
        /* lock wasn't in chain - don't drop through to free memory */
        {
        Permit();
        BUG(("Lock not found in chain\n"));
        return;
        }
     else
        this->fl_Link = lock->fl_Link;
     }
  Permit();

  DosFreeMem((char *)lock);
}

void ActLock(global, pkt)
GLOBAL global;
struct DosPacket      *pkt;      /* a pointer to the dos packet sent       */
/* Arg1: Lock */
/* Arg2: Name */
/* Arg3: Mode: ACCESS_READ, ACCESS_WRITE */
{
  KEY key;
  struct FileLock *lock;

  BUG(("Action Lock\n"));
  BUGBSTR("File to lock: ", pkt->dp_Arg2);

  key = GetKey(global, (struct FileLock *)pkt->dp_Arg1);
  key = LocateEntry(global, key, (char *)pkt->dp_Arg2);
  lock = CreateLock(global, key, pkt->dp_Arg3);

  BUG(("Lock created at %08lx\n", lock));

  pkt->dp_Res1 = MKBADDR(lock);
}

void ActDupLock(global,pkt)
GLOBAL global;
struct DosPacket      *pkt;      /* a pointer to the dos packet sent       */
{
  struct FileLock *lock;

  BUG(("Action DupLock\n"));

  lock = (struct FileLock *)pkt->dp_Arg1;
  lock = CreateLock(global, lock->fl_Key, lock->fl_Access);
  pkt->dp_Res1 = MKBADDR(lock);
}

void ActUnLock(global, pkt)
GLOBAL global;
struct DosPacket      *pkt;      /* a pointer to the dos packet sent       */
{
  struct FileLock *lock;

  lock = (struct FileLock *)pkt->dp_Arg1;

  BUG(("Action UnLock\n"));

  freelock(global, lock);
  pkt->dp_Res1 = DOS_TRUE;
}
```

Programming In M2AMIGA

Interfacing with Assembly, Part I

by Anthony Bryant
Copyright © 1988 A. Bryant

Anthony Bryant is a computer engineer working with remote systems and robotics. He programs the Amiga in his spare time. This article is part of a series of articles on using the Modula 2 programming language, specifically the M2AMIGA Development System from Interface Technologies.

Sooner or later, for reasons of speed, space optimization or complexity, the M2AMIGA programmer finds the need to augment his M2 library with some low level assembly language routines.

While Amiga's *Exec* supports high level language interfacing to many facets of the Amiga hardware, if you have to interface with a device handler or an interrupt server, say, or any routine that is one of possibly many chained system routines in the multi-tasking Exec, then your routines must be as efficient as possible. This usually means using assembly language.

Perhaps a programming problem involves a time-critical procedure, or you want to do something special (or nasty) with input event handling or text output processing. Perhaps there is a super-fast, fixed-point math package you've been wanting to try out. There are times when you need to know how to 'glue' assembly language routines onto higher level language programs.

Interfacing with internal (the Amiga Library) and external (roll-your-own) routines is the subject of this article. I readily admit to a certain longtime interest in language implementations, and I admit to also being curious to see what style of code this M2AMIGA compiler generates! Alright, call me nosey, then.

It is also interesting to see how the different high-level language control statements are constructed in assembly, and from this, how to improve one's coding techniques in M2. Most assembly programmers know never to multiply or divide by a power of two when you can shift instead. A shift is also faster in M2AMIGA.

M2AMIGA has standard procedures REG(), SETREG() and INLINE() with which to get or set (data or address) register values, or insert machine code (16-bit words). However, it is also instructive to know how to pass parameters to your own assembly procedures. This lets you use the speed and compact code of assembly language together with the ease of use of a high level language like M2.

Every compiler writer imparts his own particular style to the code generated, but primarily, it is from the language itself that emerges an overused small subset of instructions. Every language, too, may use a different sub-set. Even the assignment of registers may differ from one compiler to another, until you get down to the underlying assembly language conventions on the Amiga.

The Task Stack

Every task in the Amiga requires a stack. All task stacks are *user mode* stacks (in the language of the 68000) and are addressed through the A7 register. All normally executing code uses this task stack. System interrupts and other low-level operations use a special *supervisor mode* stack and do not directly affect task stacks. (When a task is interrupted, all of its current registers are preserved on the supervisor stack.) Task stacks are thus normally only used to store local variables, subroutine return addresses and saved registers.

The M2 programmer is blissfully unaware of the stack, but, as in other high-level languages, the compiler needs to be concerned with all code, data and stack space. The code needs to be able to access the data and stack space, and recursive code is a heavy user of the stack. Because stack checking is not provided as a service of Exec, the M2AMIGA compiler does stack overflow checking as a preamble to every procedure call to see if sufficient stack space is available for its needs (i.e. parameter passing and room for local variables). The compiler can be instructed to switch off stack checking and avoid putting in the extra code.

Parameter passing: stack it!

In M2, as in other high-level languages, parameters are normally passed to procedures via the stack in a standard way.

This is normally called the *stack protocol* or *calling sequence*, but in M2 lingo is known as the *activation record*. For every procedure call M2 makes, an activation record is placed on the stack. Parameters are pushed on the stack in left-to-right order, and then the procedure is called, pushing the return address on the stack. (Note that in the C language, parameters are pushed on the stack in opposite order: right to left.)

For example, suppose the following procedure definition:

```
TYPE proc = PROCEDURE (ADDRESS, INTEGER, CARDINAL );
```

Then to call this procedure by value - proc(#addr, #int, #card) the compiler generates:

```
MOVE.L  #addr, -(A7); push the first value parameter
MOVE.W  #int,  -(A7); push the next value parameter
MOVE.W  #card, -(A7); push the last value parameter
BSR     proc        ; call the procedure
```

Control is then transferred to the procedure that is responsible for maintaining a proper activation record. Certain registers have been assigned or reserved for specific purposes relating to this. Register A6 is reserved for use as the *universal* library base pointer. Register A5 is the procedure's local *stack frame pointer* used to access the local variables area, within the stack. Register A4 is used to point to the global variable area, within the program space, and is often referred to as the *base address* pointer.

Procedures normally start with a chunk of entry code that fetches the base address pointer and allocates space on the stack (with a **LINK** instruction) for local variables, if there are any. The old frame pointer and base address pointer are saved before new ones are set up. Recursive procedures thus have a new stack area with new local variables each time they are called.

The base pointer indexed addressing mode is equivalent to the BASEREG A4 option available on some Assemblers in conjuction with the SMALLDATA option of the *BLink* linker. Because all code must be able to access global data from this base address, the 16 bit relative addressing mode used restricts the size of code and global data area to 32k per module. If you have a huge amount of data, you need to split up your program code and data into separate modules. (This is another good reason to modularize your program.) Registers A4 and A5 are initially set up at the start of every module.

Assuming that our procedure has no local variables, here is the entry code, generated by the compiler:

```
MOVE.L A4, -(A7) ; push A4, set global data pointer, A4
MOVE.L  -gdata(PC), A4 ;  to point to global data area

                 ; (no local variables, so..)
MOVE.L  A5, -(A7) ; push A5, and set frame pointer, A5
MOVE.L  A7, A5    ;  to point to top of stack
```

Now our procedure needs to get at values that were pushed on the stack. Because you can't do much with them stuck on the stack, you want these values copied to registers. To calculate the stack offsets, you have to know the size of each parameter (if they were pushed on the stack with a MOVE.L or a MOVE.W instruction), and you have to know what else is on the stack! When a procedure is entered and a new stack frame initialized, A5 always points to just below the return address and saved global data pointer, i.e. at the saved old A5 value.

Currently the stack frame (activation record) looks like this:

```
#addr          at offset 16
#int           at offset 14
#card          at offset 12
return address at offset 8
A4             at offset 4
A5             at offset 0
```

The new frame pointer, A5, points to the current top of the stack. So... some hand-coding to move to registers, A1,D0,D1:

```
MOVE.L  16(A5),A1 ;first parameter on stack is #addr
MOVE.W  14(A5),D0 ; next parameter on stack is #int
MOVE.W  12(A5),D1 ; last parameter on stack is #card
```

Normal assembly code register conventions dictate that only D0, D1, A0 and A1 are free for scratch use. In M2AMIGA procedures, data register D0 is used to return integer values (as required in Amiga Library routines, for example) and address register A0 is often used as a fast subroutine call return. (Although **RTS** provides a simple return mechanism, it is faster, especially for frequently called subroutines, to store the return address in a register, like A0, and then **JMP (A0)**). But you can still use A0 for an input parameter, as long as it doesn't need to be preserved.

It may be unwise to assume too much when it comes to 'borrowing' other registers, especially if you intend to interface with system routines where registers may be preset with various values that must not be modified. If in doubt, it's a good habit to save any registers used within an assembler routine before calling the routine, and restore them afterwards.

Procedures normally end with a chunk of exit code that restores the stack and registers A4 and A5. In M2, just as in assembly language, it's the procedure's responsibility to clean up the stack. (Note that in the C language it's the caller, not the subroutine, that cleans up the stack.)

Do you remember what was on the stack? Well, the compiler does; it generates:

```
UNLK    A5              ; unlink stack and pop A5
MOVE.L  (A7)+, A4       ; pop A4
MOVE.L  (A7)+, A0       ; pop return address into A0
ADDQ.L  #params,A7      ; deallocate <params> bytes
JMP     (A0)            ; return
```

(In this example, *params* = 8 (bytes); a long and two words were previously pushed).

The 68000 stack grows downward, (towards lower memory addresses), so to deallocate space or to just discard previously pushed values, you add to the stack pointer. Small adjustments can be made to the stack pointer with ADDQ or SUBQ, but these 'quick' instructions can change it by at most 8 bytes, so the instruction:

```
LEA params(A7), A7
```

may be generated, instead.

The compiler can be instructed not to generate the deallocation code, for special instances. In such a case the exit code simplifies to:

```
UNLK    A5              ; unlink stack and pop A5
MOVE.L  (A7)+, A4       ; pop A4
RTS                     ; return
```

Returning parameters

For function procedures, parameters are returned in data registers. Normally, D0 returns the four-byte result, but help from D1 may be required if eight bytes (for example, a LONGREAL) are returned.

Up to now we have only considered passing *call by value* parameters to a proper procedure. But proper procedures may also return parameters using the VAR type, or *call by reference*, technique on variables. In this case, it is the address of the variable that is pushed on the stack, and it's the same address that is poked with a return value.

Parameter passing: register it!

There is another, easier, way. It avoids having to remember the calling sequence - exactly what was pushed and by how many bytes - and perhaps more importantly, communicates to program readability by telling the compiler exactly what you intend to do.

M2AMIGA enables parameters to be passed via specific registers. Devised especially for Amiga library procedures, which require that parameters be passed in registers, this technique can also be used with your own procedures.

Address register parameters are represented by the numbers 8 to 15 and data registers by the numbers 0 to 7, enclosed in braces following the name of the formal parameter.

Now, if we change our procedure definition:

```
TYPE  proc = PROCEDURE (ADDRESS {9},INTEGER {0},CARDINAL {1});
```

Calling this procedure by value effectively generates:

```
MOVE.L #addr,A1 ;transfer the first parameter to A1
MOVE.W #int,D0  ;transfer the next  parameter to D0
MOVE.W #card,D1 ;transfer the last  parameter to D1
BSR    proc     ; call the procedure
```

Procedure entry code is generated as before, allocating space on the stack for local variables, if any, and preserving A4 and A5, but now we don't have to look on the stack for our values, since they are already in the registers we've assigned. Exit code simplifies to:

```
UNLK    A5              ; unlink stack and pop A5
MOVE.L  (A7)+ , A4      ; pop A4
RTS                     ; return
```

With parameters passed via registers, no extra stack work is required. You still have to be careful not to trash registers already in use, but this is now identical to the exit code for parameter deallocation. For a lot of assembly language applications that, say, operate on global variables or need only to save registers on the stack (and therefore clean up after themselves), it's just what you need. Stack space is also often used for temporary structures and arrays.

The M2AMIGA functions, SETREG() and REG() allow for the direct intermixing of M2 with assembly code. Use SETREG() to transfer (copy) a given variable into a given register in order to access local or global variables. The REG() function can be used to return the value of a given register back to a variable.

For example, within a procedure:

SETREG(9, a); generates:

```
MOVE.L 16(A5),A1  ; retrieve an address, a, off the local
                  ; stack and store it in register A1 {9}
```

SETREG(6, i); generates:

```
MOVE.L -4(A4),D6 ; store global variable, i, in data
                 ; register D6 {6}
```

j := REG(0); generates:

```
MOVE.L   D0 , -8(A4) ; copies register D0 {0} to global
                     ; variable, j
```

If you don't require access to global or local data, through A4 and A5, (Don't trash these registers!), the M2AMIGA compiler can be instructed not to generate the two procedure code chunks for entry and exit.

Assembly procedures crafted with INLINE() can have an even simpler exit:

```
RTS                 ; return
```

Within M2AMIGA's Arts module, for example, there are four 32-bit, signed and unsigned, multiply and divide arithmetic routines that were crafted directly with INLINE() assembly code. Math routines are prime candidates for a need for speed! Designed especially for compiler support, these math routines can alo be used as external direct calls; the procedure definitions are clearly accessible and are, basically, all that one needs to know how to use them.

Interfacing with internal routines

Passing parameters in registers is necessary when communicating with Amiga's system routines, internally. For reasons of speed, they have been designed with assembly language calling conventions (i.e. registers) in mind. When it needs to call an Amiga Library routine, the M2AMIGA compiler loads the parameters into the right registers (in the right order), gets the right library base pointer and executes the call by generating all neccessary code inline, just like an Assembler.

```
                    ; load various registers, then
MOVE.L -xxx(PC), A6
                    ; get library base as a relative address
                    ;  offset (and relative to this MODULE)
JSR     -lvo(A6) ; call library rtn with vector offset
```

Register A6, then, is primed before every library call and so the needed library base vectors must be present in every module (actually M2AMIGA imports library names and computes library base vectors when it opens each library). M2AMIGA gleans the required registers' parameter assignments, relating to library routines, and the library vector offsets, from the imported symbol file. Passing parameters using this method, rather than the stack method, eliminates the need for the extra 'stub' binding routines required in C.

Communication works the other way also, when internal system routines are required to call your own 'handler' or 'server' code. For example, no additional assembly code is needed (to convert the calling sequence) when the input device stream handler calls your own special procedure. The system expects to pass (via registers) to your high-level language procedure, 1. a pointer to an InputEvent in A0, and 2. a pointer to a data area in A1.

Set up your procedure definition to accomplish this directly with:

```
PROCEDURE myHandler(iev{8}: InputEventPtr; myData{9}: ADDRESS);

                    A0                    A1
```

Of course, you also need to know how this system routine behaves before you can complete your myHandler() routine. This will determine whether or not you need to access global variables and thus need the normal entry/exit code and whether the bulk of the routine can be written in M2 or assembler. Keep that ROM Kernal manual handy, too!

Interfacing with External Routines

The secret of fast and compact code on the 68000 is to make use of registers. If you keep variables in registers, rather than in memory, and perform arithmetic operations on registers, rather than on memory, then you can achieve a significant saving in time and code.

For the most part a compiler also tries to adhere to these precepts, keeping the most used variables in registers. It's all part of the optimizing process, included in a compiler's automatic generation of code. But a compiler also has to oversee runtime range checks, division by zero and overflow / underflow checks after every operation. If you have tested your program carefully, you may wish to switch off these checks in order to gain a little more speed and eliminate the extra code.

The ability to move from high-level to low-level routines and back is made much easier if you are aware of stack protocol. While 'activation records' and call-by-whatever might seem confusing at first, once you know the protocol, the whole process becomes almost transparent. M2AMIGA offers programmers a choice in passing parameters that allows greater flexibility and expands the use of a high-level language.

Next month, we'll grapple with a practical example program, and disassemble some code.

3D OPTIONS

CONVERT IFF BITMAPPED PICTURES TO 2D AND 3D OBJECTS!!!

Now You Can Make Use of All Those DPaint and Digitized Pictures in Your Favorite 3D or CAD Package

VIDEOSCAPE 3D, AEGIS DRAW, INTROCAD, MCAD, PROFESSIONAL PAGE, POSTSCRIPT DXF (AUTOCAD...) & More.

TO ORDER CALL
1 800-628-2828 Ext 829

We accept VISA, MASTER CARD and AMERICAN EXPRESS

Only $49.95

Or Send Check or Money Order to:
Rainbows Edge Productions
4412 4th Avenue Suite 2
Brooklyn, NY 11220

All product names are Trademarks of their respective companies

AmigaDOS Patterns In Modula-2

The theory and practice of file name patterns

by Clark 'Top Gun' Williams

Whenever computer science linguists begin the definition of a new language, they are faced with a number of decisions. A big one is choosing the symbols used to represent the language, as well as designing its grammar and syntax. In AmigaDOS wildcarding we use the special characters **?**, **%**, **#**, (,), ' and |, as well as any character that can be part of a legal AmigaDOS file name.

These characters actually represent a way to communicate to (you might even say, "program file names in") AmigaDOS. Basically there are two kinds of characters in this symbol set. There are match characters: **?**, **%** and <any-character>. And there are control characters: ', **#**, (,), |. By appropriately grouping and ordering these characters, we can use them to represent a pattern that may be 'recognized' by a program.

Naming conventions

A developer should make every attempt to name files generically. For example, the source code to a Modula-2 program that controls a PROM burner might have file names such as *Promxxx.DEF* or *Promxxx.MOD*, where the *xxx* is some arbitrary string of characters (e.g. **PromInit.DEF**, **PromGlobals.DEF**, **PromWrite.MOD**).

The **DEF** and **MOD** parts of the names are called 'file extensions'. Extensions were originally used as a tool for programmers to help order their files. Conventional extensions such as **BAS** for **BASIC** files and **PAS** for Pascal files provide a standardized way of using the name of the file to specify its type. We owe the fact that file name extensions most commonly have three characters to a historical accident: this particular format happened to suit the architecture of systems on which extensions first became widespread, such as Digital Equipment Corporation's PDP series.

Having once adopted the file-naming structure of *<name>.<ext>*, the next logical step is to devise a tool for manipulating these file names - wildcarding.

Wildcarding

Wildcarding allows a programmer to refer to all those files that begin with *<name>* and end with a period followed by three characters of any kind. For example, we would like to be able to specify a pattern that refers to all file names beginning with **PromInit.**, as long as that string is followed by three characters of some kind.

The kind of symbol we need is called a 'don't care' indicator. In hardware, a bit whose value is of no importance is usually referred to as a 'don't care bit', and is indicated by an **X**.

What character shall we pick to mean 'don't care'? An obvious choice for the purpose is the question mark. A **?**, then, means: "Match any single character at this point." The wildcard pattern *PromInit.???* therefore matches the file names **PromInit.DEF**, **PromInit.MOD**, **PromInit.DAT**, **PromInit.ABC**, **PromInit.123**, etc. A wildcard pattern of *Prom????.???* could give you an enormous number of matched file names.

What about the very similar file name **PromInit2.DEF**? It will *not* be matched - and suddenly the problem of wildcarding has become more complicated. Since a **?** character only matches a single character, the pattern *Prom????.???* will match **PromInit.DEF**, but **PromInit2.DEF** will not be matched because of the extra character in **Init2**. If you try to get around this by adding another **?**, you now won't match *Init*, because **?** says that there *must* be a character to match against. With five **?**s and four characters there is no match.

A partial solution is to include a new wildcard character: **%**. This character matches any character (just as **?** does), but also matches the null character! Note that introducing **%** does not make the **?** obsolete. For instance, consider the difference between the patterns *Prom????%.D??*, which will match 'Prom' files having either four or five supplementary characters; and the pattern *Prom%%%%%.D??*, which would match 'Prom' files having from zero to five extra characters.

We now have two characters in our pattern-generating tool kit: **?** and **%**... or do we?

Actually, you may have noticed a very subtle thing happening. We really have *three* characters in our kit. The **.** ('dot') character has been acting as an alignment character (a 'control' character). The dot is a special character: it aligns the extension to the pattern. This is *not* a good thing!

In the world of PDP 11 file names, the dot *is* an alignment tool - it is not a part of the file name per se. File names are stored in a fixed width format padded with blanks. The extension is stored following the blanks and without the dot character. Under other file naming conventions, though, the dot *is* stored as part of the file name. This is a better approach: The dot is a perfectly reasonable separator, but so too are the space, the comma and the dash. One should be able to use whatever extension and extension designator one wishes. Attempting to preserve simplicity and elegance, let us not force an arbitrary choice of extension designator on the user, but instead allow the use any legal file name character. In AmigaDOS we do not use an extension designator as an alignment tool - we treat it as a normal character to be matched. The designator does not have special status or receive special handling during wildcarding.

Our example pattern *Prom????%.???* will now fail for a different reason. **PromInit.DEF** will match character by character, up to the point where the **%** matches the dot, but fail there attempting to match the **D**. Clearly there is a need for a special character that will match as many characters as there are, including no characters at all. We'll use the **#** (number) character to indicate zero or more characters. The pattern *A#B* would match:

```
A     zero occurrences of B
AB    one
ABB   two
ABBB  three
```

and so on.

Combining **#** with **?** gives us the sentence: "Match as many of *any* characters as there are". We now may write *Prom#?.???* as our pattern; this will match both **PromInit.DEF** *and* **PromInit2.DAT**.

Some operating systems have bastardized this concept by using * to mean the same thing as **#?**. This was not the original use or meaning of * in wildcarding.

The reason for using * is a result of the format of certain commands that you normally would use wildcarding with. An example is the **copy** command. How many times have you wanted to copy from *Prom#?.DEF* to some directory using the same file name as the file you are copying? The problem is that you're already in the directory into which you want to copy the files. In AmigaDOS, you will have to copy to the explicit path name of the directory you are in, in order to get everything to work. If you are deep in a tree, that path may be very long to type. You may instead use **CD /** to move up to the parent directory, then CD back down after the copy. But wouldn't it be easier to be able to copy from some directory/volume and indicate that you would like to use the same file name as the file name you are copying? That was the original use of the *. It meant 'this file'. So:

```
1>COPY DF0:Prom#?.DEF TO *
```

would be the equivalent of:

```
1>COPY DF0:PromInit.DEF TO CurrentDir/PromInit.DEF
1>COPY DF0:PromInit2.DEF TO CurrentDir/PromInit2.DEF
```

and so on.

Right, wrong or indifferent, * in AmigaDOS means 'active console', not 'this file', so we can't use * in file name patterns without overloading its meaning, and overloading in anything is a fundamental no-no! My own favourite choice for a 'this file' character is **!** (sometimes called the 'bang' character). Since this character doesn't currently have any special meaning to AmigaDOS, its use would cleanly solve the problem of copying into the current directory.

The ARP (AmigaDOS Replacement Program) folks have used * as an equivalent for **#?**. This is not the best decision in my opinion. I suppose the reason for the choice is that * has been used to mean 'any number of characters' on a few operating systems. Just because other systems have allowed themselves to degenerate into employing vague operators, though, doesn't mean that AmigaDOS has to follow the same path. To use the * for both current console and as a synonym for **#?** is unnecessary operator overloading. It also yields poor visual acuity and makes for a large headache when trying to parse things correctly. The **#?** notation is actually very elegant when you understand the meaning of the individual characters; the sentence it forms is obvious and presents no particular parsing problems. The argument that it is easier to type * than to type **#?** is perhaps the ultimate in decadence. "Learn to type!" is the proper response to such poor reasoning. The exception processing to account for * overloading requires additional code that is just not necessary. It certainly doesn't meet the desirable design objectives of simplicity and elegance!

The next problem is the monadic nature of the **#** character. ('Monadic'... Sounds *great*, doesn't it? It just means that **#** only works on the very next character.) A recurring pattern such as *ABABAB* is difficult to account for without additional characters in our symbol set, so we add the grouping characters (and). For the pattern *#(AB)* we match: nil, AB, ABAB, ABABAB, and so on.

Sometimes you also run into problems interfacing to your system. In AmigaDOS commands, the space character is a significant separator. In AmigaDOS file names, though, the space is a valid, matchable character. Since **?** is also a legal character in file names (in fact, almost any character will work in AmigaDOS file names) we will add one more control character to our repertoire. The meaning of this control character will be: take the character that follows literally.

This new control character is the *squote* (single quotation) character. When scanning for the arguments supplied on a command line, AmigaDOS uses the space character as the separator character. So we must, when using AmigaDOS command lines, enclose the entire string within quotes. When

using a program, this may or may not be required depending on how the input is being done. So whereas we might have entered *Prom#? DAT*, we instead must enter the string *Prom#?' DAT* on a command line. Thus:

```
1>COPY "Prom#?' DAT" TO !
```

will match **PromInit DAT**.

Only two symbols remain to be defined. The first is the | (or) character. To match names like **AProm.MOD** and **BProm.MOD**, we can set up the pattern *(A|B)Prom#?*, which should match **AProm.MOD** or **BProm.MOD** or **AProm.DEF** or **BProm.DEF**, and so on.

Finally, there is the one character that everyone leaves out. This is the *not* operator, which should match any string *except* strings that match the pattern the operator is applied to. I propose the character ~ (the tilde) for this purpose.

Certain definitions must be made for ~; I suggest the following:

```
~?            %
~|            illegal
~<any-char>   match any-char except <any-char>
~(...)        succeed if grouped pattern fails
~#<any-char>  same as ~<any-char> (as if without the #)
~'<any-char>  same as ~<any-char> (as if without the ')
```

Wildcarding in Modula-2

One of the problems I've run into over my years of using the Amiga is trying to find a good AmigaDOS wildcarding library routine. One may in fact exist. However, if it does, it is not included in the packages available from the current suppliers of Modula-2 compilers.

You may think that the code to check the patterns you could create with the characters **?**, **#**, **|**, **%**, **'**, (,) would be quite large. Actually the question comes down to whether there is a finite state automaton that can represent the patterns and, if so, can we build a compiler for the automaton?

Well, the answer is yes to both parts of the question! (Naturally, or why write this article?) We *can* build a compiler that will generate a state machine and an interpreter that will produce the correct pattern recognition to follow the state machine.

This is well presented, using code in BCPL, in the paper "A Compact Function for Regular Expression Pattern Matching", by Martin Richards. The original paper is published by John Wiley & Sons, Ltd. The copyright is owned by John Wiley & Sons, Ltd. 0038-0644/79/0709-0527.

For Modula-2 programming, we would like to have these routines packaged into a library module. Let's develop that module now, and demonstrate how the routines may be used.

First step: Define the interface! The first action item this module must provide will check if a file name fits a wildcarding pattern. This will be called *NameFitsPattern*. It will return TRUE if the name does fit the pattern, and FALSE if either the pattern is invalid or the name does not fit the pattern. The two inputs necessary will be the wildcarding pattern and the actual file name. The single output will be the BOOLEAN value.

How shall we describe the data structure that *NameFitsPattern* will use? Let's look at what is required for the code to work. BCPL strings use a format that would appear as a record structure in Modula-2. The exact form would be this:

```
TYPE
BcplString = RECORD
Length     : BYTE;
Characters : ARRAY BYTE OF CHAR
END (* Bcpl String *);
```

I am assuming here that BYTE is defined to be a subrange of CARDINAL but doesn't take up a full 16-bit word to store an 8-bit value. This may or may not be true on your compiler; check it out first for yourself.

The fact that a BCPL string is done this way is just a fact of life. Modula-2 programmers normally don't define strings this way and we won't start insisting that they begin with this program. We will let the Modula-2 programmer continue to define arrays normally and instead we will convert the string to a BCPL string equivalent after the routines are called.

So let us ask for an ARRAY OF CHAR for both the pattern and file name values. We will ask for the ARRAY OF CHAR as a VAR parameter (to reduce the amount of overhead on the stack) and the wildcarding pattern as a call-by-value parameter. This is just in case the user wants to provide a string constant. We now have the procedure definition that we can enter into the DEFINITION MODULE. Our definition module will be called *AmigaDosWildcarding*.

The implementation of the rest of the module is fairly simple. The routine simply builds a finite state automaton (compiles the pattern) called the *Aux* array. The *Aux* array contains the information about the 'exits' - the places where control characters in the pattern are found. These are the places where you would stop checking - in other words, the possible exits from the checking of the pattern against a name. A recursive decent compiler called *CmplPat* sets up this *Aux* array. You'll find it in the module called *Compiler*.

Once the finite state automaton is built, you call the interpreter to scan the *Aux* and *Pat* arrays comparing the actual file name string with the state machine. If you get to the special state called *exit*, then you have finished successfully and you return TRUE. If, on the other hand, you find an error, you return FALSE.

It is important to note that the routines are written assuming that the arrays are in a BCPL string compatible format. That is,

the ARRAY OF CHAR corresponds to the record shown above. This demands some preparation work before we call the standard routines. We must first convert (transform) our pattern and name arrays from the Modula-2 character arrays we are accustomed to into the BCPL string compatible records. BCPL (and C) don't need to define a separate record, since a character string may be a numeric string or a string of ordinal values or characters or whatever and no type conflict will take place. We will fake (via type coercion) the first value of our array as having a numeric value. We will accomplish this by first moving all the characters down by one and placing the length of the string into the zeroth element of the character array. This kind of array carries the type designation of a *Chary* (CHaracter ARraY).

The *Aux* array, on the other hand, is an array of indices into the pattern that represent places where the influence of the control character of the pattern stops. These are then variables that will have a value from 0 to 255, since there can only be 255 characters in a BCPL string. This array is designated as a *Byary* (BYte ARraY).

Since we will often need a pointer to these structures, we will designate two types as pointer types to these arrays. *CharyPtr* and *ByaryPtr* will point to one of the two structures as denoted by their names.

When a pattern and a name come in, we will transform the pattern and name into BCPL string compatible structures, then call the compiler and, if there is no error, call the interpreter to see if there is a match between the file name and the wildcard pattern. This is accomplished in the IMPLEMENTATION MODULE *AmigaDosWildcarding*.

Finally, the calls to the other modules are fairly straightforward. *WildcardingGlobals.DEF* contains the global definitions used in compiling and interpreting the state machine. *WildcardingInit.MOD* sets up the pointer values, and then sets up the character arrays to look like BCPL strings. *Compiler* and *Interpreter* do the actual work in setting up our *Aux* array and scanning for a correct wildcard match.

The only other routines of interest are in the *WildcardingMisc.MOD* module. These are the utility routines needed to accomplish the compile and interpret of the wildcarding pattern.

Listing: A Modula-2 recognizer for AmigaDOS patterns

```
(*
   Copyright 1989 by Software Industry & General Hardware.
   All rights reserved.
   Author: Clark Williams

   Permission to distribute this code freely, is hereby given,
   as long as this copyright notice, author's name and the
   address below are included in the distribution.

   Software Industry & General Hardware,
   POB 1185
   Ramona, California
         92065, U.S.A.
*)

DEFINITION MODULE AmigaDosWildcarding;

(*
   Abbreviated data dictionary:

   Variable Name, is an array of characters representing an AmigaDOS file
   name and/or path.

   Variable Pattern, is an array of characters representing an AmigaDOS
   wildcard character.

   The function returns TRUE if the name supplied fits the wildcarding pattern
         supplied.  Otherwise it returns FALSE.
*)

PROCEDURE NameFitsPattern (        Name    : ARRAY OF CHAR;
                            VAR Pattern : ARRAY OF CHAR    ) : BOOLEAN;
END AmigaDosWildcarding.


IMPLEMENTATION MODULE AmigaDosWildcarding;

(*
   Abbreviated data dictionary:

   Variable Name, is an array of characters representing an AmigaDOS
   file name and/or path.

   Variable Pattern, is an array of characters representing an AmigaDOS
   wildcard character.

   Function NameFitsPattern, returns TRUE if contents of variable Name,
   supplied by calling routine, fits the wildcard pattern supplied as
   the contents of variable Pattern. Otherwise the function returns FALSE.
*)

FROM WildcardingInit   IMPORT TransformToBstr, MakeNew, CleanUp;
FROM WildcardingGlobals   IMPORT CharyPtr,   ByaryPtr;
FROM Compiler      IMPORT CmplPat;
FROM Interpreter   IMPORT Match;

PROCEDURE NameFitsPattern (       Name    : ARRAY OF CHAR;
                            VAR Pattern : ARRAY OF CHAR  ) : BOOLEAN;

VAR
 BPattern     : CharyPtr;
 BName        : CharyPtr;
 CmplPattern  : ByaryPtr;
 MatchedValue : BOOLEAN;

BEGIN (* Name Fits Pattern *)
 IF MakeNew ( BPattern, HIGH ( Pattern ), BName, HIGH ( Name ), CmplPattern )
  THEN (* no allocation of memory errors found so continue *)
   TransformToBstr ( Pattern, BPattern );
   TransformToBstr ( Name,    BName   );
   MatchedValue := FALSE;
   IF CmplPat ( BPattern, CmplPattern )
    THEN (* No error on compile of pattern, so continue. *)
     MatchedValue := Match ( BPattern, CmplPattern, BName )
   END (* IF *);
   CleanUp ( BPattern, HIGH ( Pattern ), BName, HIGH ( Name ), CmplPattern );
   RETURN MatchedValue
  ELSE (* trouble (no memory?) in paradise - leave with error. *)
   RETURN FALSE
  END (* IF *)
END NameFitsPattern;

BEGIN (* AmigaDOS Wildcarding *)
END AmigaDosWildcarding.


DEFINITION MODULE Compiler;

FROM WildcardingGlobals IMPORT CharyPtr, ByaryPtr;

PROCEDURE CmplPat ( Pattern : CharyPtr; CmplPattern: ByaryPtr ) : BOOLEAN;

END Compiler.
```

```
IMPLEMENTATION MODULE Compiler;

FROM WildcardingGlobals IMPORT EndStreamCharacter, MaxCharsInInputLine,
                               InputLineVector,    Chary,
                               CharyPtr,           Aux,
                               Ch,                 ErrFlag,
                               Pat,                PatP,
                               PatLen,             Succflag,
                               Work,               W,
                               Wp,                 Byary,
                               ByaryPtr,           Byte;

FROM WildcardingMisc    IMPORT Rch, SetExits, Exp, PutByte;

PROCEDURE CmplPat ( Pattern : CharyPtr; CmplPattern : ByaryPtr ) : BOOLEAN;

VAR
 I     : Byte;
 Temp  : Byte;
 Temp2 : Byte;

BEGIN (* Cmpl Pat *)
 Pat^ := Pattern^;
 Aux^ := CmplPattern^;
 PatP := 0;
 PatLen := Byte ( Pat^ [ 0 ] );

 ErrFlag := FALSE;
 I := 0;
 WHILE I <= PatLen DO (* stick in nulls to clear out Aux table. *)
  PutByte ( Aux, I, Byte ( EndStreamCharacter ) );
  INC ( I )
 END (* WHILE *);
 Rch;
 Temp  := 0;
 Temp  := Exp ( Temp );
 Temp2 := 0;
 SetExits ( Temp, Temp2 );
 RETURN NOT ErrFlag
END CmplPat;

BEGIN (* Compiler *)
END Compiler.


DEFINITION MODULE Interpreter;

FROM WildCardingGlobals IMPORT CharyPtr, ByaryPtr;

PROCEDURE Match ( Pat : CharyPtr; Aux : ByaryPtr; Str : CharyPtr ) : BOOLEAN;

END Interpreter.


IMPLEMENTATION MODULE Interpreter;

FROM WildCardingGlobals IMPORT MaxCharsInInputLine, InputLineVector,
                               Chary, CharyPtr, Aux, Ch, ErrFlag,
                               Pat, PatP, PatLen, Succflag, Work,
                               W, Wp, EndStreamCharacter,
                               Byary, ByaryPtr, Byte;

FROM WildcardingMisc    IMPORT Put, GetByte;

PROCEDURE Match ( Pat : CharyPtr; Aux : ByaryPtr; Str : CharyPtr ) : BOOLEAN;

(*
   Pat = Unaltered pattern.
   Aux = Auxiliary table.
   Str = String to match against the pattern.
*)

VAR
 S : Byte;
 N : Byte;
 P : CHAR;
 K : CHAR;
 Q : Byte;
 I : Byte;

BEGIN (* Match *)
 (* NEW ( W ); W is already allocated from AllocMem *)
 S := 0;
 Work^ := W^;
 Wp := 0;
 Succflag := FALSE;
 Put ( 1 );

 IF GetByte ( Aux, 0 ) <> Byte ( EndStreamCharacter )
  THEN Put ( GetByte ( Aux, 0 ) )
 END (* IF *);

 REPEAT
  N := 1;
  WHILE N <= Wp DO
   P := CHAR ( Work^ [ N ] );
   K := CHAR ( GetByte ( Pat, Byte ( P ) ) );
   Q := GetByte ( Aux, Byte ( P ) );
   CASE K OF
    "#" : INC ( P ); Put ( Byte ( P ) ); DEC ( P ); Put ( Q ) |
    '%' : Put ( Q )                                           |
    '(',
    '|' : INC ( P ); Put ( Byte ( P ) ); DEC ( P );
     IF Q <> 0 THEN Put ( Q ) END (* IF *)
    ELSE (* do nothing *)
   END (* CASE *);
   INC ( N )
  END (* WHILE *);

  IF S >= GetByte ( Str, 0 ) THEN RETURN Succflag END (* IF *);

  IF Wp = Byte ( 0 ) THEN RETURN FALSE END (* IF *);
  INC ( S );
  Ch := CHAR ( GetByte ( Str, S ) );

  (* Now deal with match items *)
  N := Wp;
  Wp := 0;
  Succflag := FALSE;

  I := 1;
  WHILE I <= N DO
   P := CHAR ( Work^ [ I ] );
   K := CHAR ( GetByte ( Pat, Byte ( P ) ) );
   CASE K OF
    '#',
    "|",
    "%",
    "(" : (* Do nothing *)                                   |
    "/" : INC ( P );
          K := CHAR ( GetByte ( Pat, Byte ( P ) ) );
          DEC ( P );
     IF Ch = K
          THEN
           Put ( GetByte ( Aux, Byte ( P ) ) )
          END (* IF *)                                      |
    "?" : Put ( GetByte ( Aux, Byte ( P ) ) )               |
    ELSE
     IF ( Ch = K )
      THEN
       Put ( GetByte ( Aux, Byte (  P ) ) )
     END (* IF *)
   END (* CASE *);
   INC ( I )
  END (* WHILE *)
  UNTIL FALSE (* do until the sun burns out! *)
END Match;

BEGIN (* Interpreter *)
END Interpreter.


DEFINITION MODULE WildcardingGlobals;

TYPE
 Byte = [ 0..255 ];

CONST
 EndStreamCharacter    = 00C;
 MaxCharsInInputLine   = Byte ( 250 );
 MaxElementInInputLine = Byte ( 249 );
```

```
TYPE
 InputLineVector = Byte;

 Chary           = ARRAY Byte OF CHAR;
 CharyPtr        = POINTER TO Chary;

 Byary           = ARRAY Byte OF Byte;
 ByaryPtr        = POINTER TO Byary;

VAR
 Aux      : ByaryPtr;

 Ch       : CHAR;

 ErrFlag  : BOOLEAN;

 Pat      : CharyPtr;
 PatP     : Byte;
 PatLen   : Byte;

 Succflag : BOOLEAN;

 Work     : ByaryPtr;
 W        : ByaryPtr;
 Wp       : Byte;

END WildcardingGlobals.


IMPLEMENTATION MODULE WildcardingGlobals;

BEGIN (* Wildcarding Globals *)
END WildcardingGlobals.


DEFINITION MODULE WildcardingInit;

FROM WildcardingGlobals   IMPORT CharyPtr, ByaryPtr;

PROCEDURE CleanUp ( VAR Pat      : CharyPtr;
                        PatSize  : CARDINAL;
                    VAR Name     : CharyPtr;
                        NameSize : CARDINAL;
                    VAR Aux    : ByaryPtr );

PROCEDURE MakeNew ( VAR Pat      : CharyPtr;
                        PatSize  : CARDINAL;
                    VAR Name     : CharyPtr;
                        NameSize : CARDINAL;
                      VAR Aux    : ByaryPtr ) : BOOLEAN;

PROCEDURE SetUpPattern ( Pattern : CharyPtr );

PROCEDURE TransformToBstr ( VAR Modula2Array  : ARRAY OF CHAR;
                                BpclStringPtr : CharyPtr        );

END WildcardingInit.


IMPLEMENTATION MODULE WildcardingInit;

FROM SYSTEM       IMPORT ADDRESS,  LONG;
FROM TermInOut    IMPORT WriteLn,  WriteString;
FROM Memory       IMPORT AllocMem, FreeMem,  MemReqSet,
                         MemFast,  MemClear;
FROM WildcardingGlobals IMPORT CharyPtr, ByaryPtr, MaxCharsInInputLine,
                               Byte,     Work,     W;

CONST
 NULL = ADDRESS ( 0 );

PROCEDURE MakeNew ( VAR Pat        : CharyPtr;
                        PatSize    : CARDINAL;
                    VAR Name       : CharyPtr;
                        NameSize   : CARDINAL;
                    VAR Aux        : ByaryPtr  ) : BOOLEAN;

CONST
 StdMemSet = MemReqSet { MemFast, MemClear };

VAR
 LongTemp : LONGCARD;
 Success  : BOOLEAN;

BEGIN (* Make New *)
 (* Initialize all the pointers to a known value. *)
 Pat := NULL; Name := NULL; Aux := NULL; Work := NULL; W := NULL;

 Pat := AllocMem ( LONGCARD ( LONG ( PatSize ) ), StdMemSet );
 IF Pat <> NULL
  THEN
   Name := AllocMem ( LONGCARD ( LONG ( NameSize ) ), StdMemSet );
   IF Name <> NULL
    THEN
     Aux := AllocMem ( LONGCARD ( LONG ( PatSize ) ), StdMemSet );
     IF Aux = NULL
      THEN (* Error on allocation. *)
       WriteString ( "Not enough memory for auxiliary buffer." );
       WriteLn
     END (* IF *)
    ELSE (* Error on allocation. *)
     WriteString ( "Not enough memory for 'name' buffer." );
     WriteLn
   END (* IF *)
  ELSE (* Error on allocation. *)
   WriteString ( "Not enough memory for pattern buffer." );
   WriteLn
 END (* IF *);

 Work := AllocMem ( LONGCARD ( LONG ( PatSize ) ), StdMemSet );
 W    := AllocMem ( LONGCARD ( LONG ( PatSize ) ), StdMemSet );

 IF Work = NULL
  THEN (* Error on allocation *)
   WriteString ( "Not enough memory for 'Work' buffer." );
   WriteLn
 END (* IF *);

 IF W = NULL
  THEN (* Error on allocation *)
   WriteString ( "Not enough memory for 'temp' buffer." );
   WriteLn
 END (* IF *);

 RETURN     ( Pat  <> NULL )
   AND ( Name <> NULL )
   AND ( Aux  <> NULL )
   AND ( Work <> NULL )
   AND ( W    <> NULL )
END MakeNew;

PROCEDURE CleanUp ( VAR Pat    : CharyPtr;
        PatSize    : CARDINAL;
         VAR Name    : CharyPtr;
        NameSize : CARDINAL;
         VAR Aux    : ByaryPtr  );

VAR
 LongTemp : LONGCARD;
 Success  : BOOLEAN;

BEGIN (* Clean Up *)
 IF Pat  <> NULL THEN FreeMem ( Pat,  LONGCARD ( LONG ( PatSize  ) ) ) END;
 IF Name <> NULL THEN FreeMem ( Name, LONGCARD ( LONG ( NameSize ) ) ) END;
 IF Aux  <> NULL THEN FreeMem ( Aux,  LONGCARD ( LONG ( PatSize  ) ) ) END;
 IF Work <> NULL THEN FreeMem ( Work, LONGCARD ( LONG ( PatSize  ) ) ) END;
 IF W    <> NULL THEN FreeMem ( W,    LONGCARD ( LONG ( PatSize  ) ) ) END
END CleanUp;

PROCEDURE SetUpPattern ( Pat : CharyPtr );

VAR
 Index : Byte;

PROCEDURE MovePatternDownByOne ( Index : Byte );

VAR
 LoopCounter : CARDINAL;
```

```
BEGIN (* Move Pattern Down By One *)
 IF ( Index > 0 )
  THEN (* *)
   FOR LoopCounter := Index TO 0 BY -1 DO
    Pat^ [ LoopCounter + 1 ] := Pat^ [ LoopCounter ]
   END (* FOR *)
 END (* IF *)
END MovePatternDownByOne;

BEGIN (* Set Up Pattern *)
 Index := 0;
 WHILE ( Index < MaxCharsInInputLine ) AND ( Pat^ [ Index ] <> 00C ) DO
  INC ( Index )
 END (* WHILE *);

 IF ( Index > 0 )
  THEN
   MovePatternDownByOne ( Index );
   Pat^ [ 0 ] := CHAR ( Index )
 END (* IF *)
END SetUpPattern;

PROCEDURE TransformToBstr ( VAR Modula2Array  : ARRAY OF CHAR;
           BcplStringPtr : CharyPtr       );
VAR
 Index          : CARDINAL;
 StopEverything : BOOLEAN;

BEGIN (* Transform To Bstr *)
 StopEverything := FALSE;
 Index := 0;
 REPEAT (* Getting characters until you find a null or end of array. *)
  (* Look for end of array. *)
  StopEverything :=    ( Index >= HIGH ( Modula2Array   ) )
         OR ( Index >= HIGH ( BcplStringPtr^ ) );

  IF NOT StopEverything
   THEN (* we are not at end of array look for null terminator. *)
    StopEverything := Modula2Array [ Index ] = 00C
  END (* IF *);

  IF NOT StopEverything
   THEN (* not at end of array and not at a null, move character. *)
    BcplStringPtr^ [ Index + 1 ] := Modula2Array [ Index ]
  END (* IF *);

  INC ( Index ) (* do next character. *)
 UNTIL StopEverything;

 (* Now set up the length in first byte and we are ready to for action. *)
 SetUpPattern ( BcplStringPtr )
END TransformToBstr;

BEGIN (* Wildcarding Init *)
END WildcardingInit.

DEFINITION MODULE WildcardingMisc;

FROM SYSTEM IMPORT ADDRESS;
FROM WildcardingGlobals IMPORT Byte, ByaryPtr;

PROCEDURE Rch;

PROCEDURE NextItem;

PROCEDURE Prim () : Byte;

PROCEDURE Put ( N : Byte );

PROCEDURE SetExits ( VAR List : Byte; VAR Val : Byte );

PROCEDURE Join ( VAR A : Byte; VAR B : Byte ) : Byte;

PROCEDURE Exp ( VAR AltP : Byte ) : Byte;

PROCEDURE PutByte ( VAR StringToUse : ADDRESS;
                        Position    : Byte;
                        Character   : Byte      );

PROCEDURE GetByte ( StringToUse   : ADDRESS;
                    PositionToUse : Byte      ) : Byte;
END WildcardingMisc.

IMPLEMENTATION MODULE WildcardingMisc;

FROM SYSTEM IMPORT ADDRESS;

FROM Strings            IMPORT StringLength;

FROM WildcardingGlobals IMPORT MaxCharsInInputLine, InputLineVector, Chary,
                               CharyPtr, Aux, Ch, ErrFlag, Pat, PatP,
                               PatLen, Succflag, Work, W, Wp,
                               EndStreamCharacter, Byary, ByaryPtr, Byte;

PROCEDURE PutByte ( VAR StringToUse : ADDRESS;
                        Position    : Byte;
                        Character   : Byte      );

VAR
 ActualString : ByaryPtr;

BEGIN (* Put Byte *)
 ActualString := ByaryPtr ( StringToUse );
 IF ( MaxCharsInInputLine >= Position )
  THEN ActualString^ [ Position ] := Character
 END (* IF *)
END PutByte;

PROCEDURE GetByte ( StringToUse   : ADDRESS;
                    PositionToUse : Byte      ) : Byte;

VAR
 ActualString : ByaryPtr;

BEGIN (* Get Byte *)
 ActualString := ByaryPtr ( StringToUse );

 IF MaxCharsInInputLine <= PositionToUse
  THEN RETURN Byte ( EndStreamCharacter )
  ELSE RETURN ActualString^ [ PositionToUse ]
 END (* IF *)
END GetByte;

PROCEDURE Rch;

BEGIN (* Rch *)
 IF PatP >= PatLen
  THEN Ch := EndStreamCharacter
  ELSE
   INC ( PatP );
   Ch := CHAR ( GetByte ( Pat, PatP ) )
 END (* IF *)
END Rch;

PROCEDURE NextItem;

BEGIN (* Next Item *)
 IF Ch = "'" THEN Rch END (* IF *);
 Rch
END NextItem;

PROCEDURE Prim () : Byte;

VAR
 A    : Byte;
 Op   : CHAR;
 Temp : Byte;

BEGIN (* Prim *)
 A := PatP;
 Op := Ch;
 NextItem;
 CASE Op OF
  EndStreamCharacter : IF PatP < PatLen THEN ErrFlag := TRUE END (* IF *) |
      ")",
      "|" : ErrFlag := TRUE                                              |
      "#" : Temp := Prim (); SetExits ( Temp, A )                        |
      "(" : A := Exp ( A );
            IF Ch <> ")" THEN ErrFlag := TRUE END (* IF *);
            NextItem ()                                                  |
   ELSE (* Do nothing *)
  END (* CASE *);
  RETURN A
END Prim;
```

```
PROCEDURE Put ( N : Byte );

VAR
 I : Byte;

BEGIN (* Put *)
 IF N = Byte ( 0 )
  THEN Succflag := TRUE
  ELSE
   I := Byte ( 1 );
   LOOP
    IF ( I > Wp )
     THEN EXIT
     ELSE
      IF ( Work^ [ I ] <> N )
       THEN INC ( I )
       ELSE EXIT
      END (* IF *)
    END (* IF *)
   END (* LOOP *);

   INC ( Wp );
   Work^ [ Wp ] := N
 END (* IF *)
END Put;

PROCEDURE SetExits ( VAR List : Byte; VAR Val : Byte );

VAR
 A : Byte;

BEGIN (* Set Exits *)
 WHILE List <> Byte ( 0 ) DO
  A := GetByte ( Aux, List );
  PutByte ( Aux, List, Val );
  List := A
 END (* WHILE *)
END SetExits;

PROCEDURE Join ( VAR A : Byte; VAR B : Byte ) : Byte;

VAR
 T : Byte;

BEGIN (* Join *)
 T := A;
 IF A = Byte ( 0 ) THEN RETURN B END (* IF *);
 WHILE GetByte ( Aux, A ) <> Byte ( 0 ) DO
  A := GetByte ( Aux, A )
 END (* WHILE *);
 PutByte ( Aux, A, B );
 RETURN T
END Join;

PROCEDURE Exp ( VAR AltP : Byte ) : Byte;

VAR
 A     : Byte;
 Exits : Byte;

BEGIN (* Exp *)
 Exits := Byte ( 0 );
 LOOP
  A := Prim ();
  IF    ( Ch = "|" )
     OR ( Ch = ")" )
     OR ( Ch = EndStreamCharacter )
   THEN
    Exits := Join ( Exits, A );
    IF Ch <> "|" THEN RETURN Exits END (* IF *);
    PutByte ( Aux, AltP, PatP );
    AltP := PatP;
    NextItem
   ELSE
    SetExits ( A, PatP )
  END (* IF *)
 END (* LOOP *)
END Exp;

BEGIN (* Wildcarding Misc *)
END WildcardingMisc.
```


MOUSE MASTER

Now For The Amiga!

Are you tired of fumbling under or behind your computer to swap your mouse and joystick cables? Are your cable and computer connectors worn out from all the plugging and unplugging? Then Mouse Master is a must for you!

$39.95
plus shipping & handling.

(602) 322-6100

Practical Solutions®
1135 N. Jones Blvd., Tucson, AZ 85716

ANNOUNCING!!

THE SERIAL SOLUTION
a dual-port internal serial card for the Amiga 2000

- one 25-pin Amiga-compatible serial port
- one 9-pin AT-compatible serial port
- supports printers, MIDI interfaces, sound samplers, drawing pads, and other serial devices
- works with small patch to most existing software
- compatible with SER: and "serial.device"

See us at World of Commodore booth 820.

Dealer Inquiries Welcome

CHECKPOINT
TECHNOLOGIES
P.O. Box 2035 • Manassas, VA 22110 • 703-330-5353

Dynamics For Animators, Part 1

Adding realism to 3D animations

by Cathryn and Eric Graham

Eric and Cathryn have been involved with three dimensional modelling on the Amiga for several years, producing the Juggler Demo, Sculpt 3D and Sculpt Animate, as well as magazine articles, and the Ray Tracing Newsletter. They live in a high mountain valley in Colorado. They can be reached at P.O. Box 8298, Durango, Colorado 81301.

The colorful spinning 'Boing Ball' falls from the sky, building up speed before it hits the ground. At the moment of impact the ball flattens to half its height and then springs back into the air, visibly quivering with the shock of impact.

In this article and the next we will show you how to calculate the motion of spinning, colliding elastic objects so that you can add a new element of realism to your three dimensional animations. Most of the great Amiga animations have disregarded the science of dynamics altogether, relying on the animator's skill to make the motions look realistic. One of the best examples is 'The Dream goes Berserk!' by Leo Schwab, which portrays a Boing Ball juggling three unicycles. Each time a unicycle hits the ball, the ball deflects in a convincing way, adding greatly to the sense of reality. Something is wrong, however, as each unicyle reaches the peak of its trajectory. The tumbling unicyles spin a little bit too vigorously, as though some unseen hand has given them an extra tweak.

The human eye and brain, after many years of observing how things move in our world, reacts strongly to anything that behaves unnaturally. Conversely, if the dynamics of an animation are done correctly, the animation will seem far more realistic. This is not to say that we must be bound by the laws of physics that apply in our universe. As an artist and a builder of universes, you are free to make your own rules, but if you stick to your rules in a consistent fashion you will be rewarded with realistic looking animations.

In this crash course we must deal with Calculus, Newtonian Dynamics and Numerical Analysis, but if your memories of these subjects are dim (or non-existent), do not worry because your computer will be doing all the hard work.

Calculus

Calculus is a branch of mathematics that deals with how one thing changes with respect to another. In analyzing motion we are concerned with how an object changes its location with the passage of time. Imagine a car moving along a straight road. The position of the car can be uniquely specified by a single number called a coordinate, which could indicate, for example, how far the car is from home. Let us call this number X. If the car moves, the value of X will be different at different times. Call the time t. The velocity of the car, call it V, is the rate at which X changes. We can write it as:

$$X' = V$$

The prime is taken to mean 'the rate at which something changes with respect to time'. This equation is called a *differential equation* and has inspired fear in generation after generation of students for the last three centuries. For example, the student could be told that the velocity is given by some complicated formula, usually involving X, and is asked to determine a formula for X. If the student is successful, he may be able to derive some insight from the formula. In real world problems, he may just be interested in evaluating his formula and will use a computer to calulate X for a number of different values of t. In that case it is often easier to let the computer evaluate the differential equation directly. This is called a numerical solution rather than an analytical solution.

It is useful to be aware of the advantages and disadvantages of both ways of solving differential equations. The biggest advantage of an analytical solution is that by looking at the formula you may be able to draw some significant conclusion: for example, X may never exceed a certain value. With a numerical solution, you can never be so sure; perhaps if you extended the solution for larger values of t, X would get bigger. These kind of insights are usually not too important for animators.

For our purposes, the biggest drawback of the quest for an analytical solution is the difficulty in finding one. Even if you

are skilled in calculus, you may find that you are unable to solve simple-looking equations. You then have two ways out. The first way is really a cheat: you define the solution to be a new function and modestly name it after yourself. The other way is hardly better: you simplify the original equation until you can solve it, and hope that the solution to the new equation bears some resemblance to the solution you wanted.

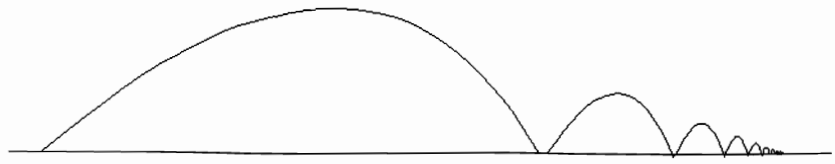


Figure 1: *particle trajectory assuming air resistance, gravity, and elasticity.*

Numerical analysis

Numerical analysis is the art of calculating approximate solutions to well-posed problems. In the days before computers, it was essential to consider the efficiency of a method, because every addition or multiplication took several seconds. Nowadays, the computation time can be so short that it is usually more important that the method be robust and reliable.

As a simple example, consider the case of an exuberant but law-abiding driver. He starts his car at time t=0, when his location is X=0. He accelerates as quickly as possible, but does not exceed the 30 MPH speed limit. After 500 feet he passes a 55 MPH sign, so he accelerates again. The problem is to find out how far he has gone after some particular time. Acceleration, call it A, is just the rate at which velocity changes, so our differential equation becomes:

$$X'' = A$$

Because the most complicated term has two primes, this is called a second order differential equation. Higher order differential equations can always be broken into a system of first order equations. In our case they are:

$$X' = V$$
$$V' = A$$

The variables that have primes next to them (X and V) are called the dependent variables. The variable with respect to which changes are measured is called the independent variable (t). When we have a system of many equations, we can store the dependent variables in an array and specify the set of equations by a single function that calculates all the primed quantities, or *derivatives* as they are called.

In C, we could write the function as follows:

```
#define FPS     1.4666666666

deriv(yprime,y,t)
double yprime[2],y[2],t;
{double A,V,X,vdesired;

 X=y[0]; V=y[1];

 if (X < 500)
    vdesired=30*FPS; /* inside 30 mph zone */
 else
    vdesired=55*FPS; /* in 55mph zone */

 A=24*(vdesired-V)/vdesired;

 yprime[0]=V;            /*  X' = V  */
 yprime[1]=A;            /*  V' = A  */
}
```

A few things are worth noting. Although the problem is posed in units of miles per hour and feet, it is best to use a single set of units. In our case we have used feet and seconds. The constant FPS is used to convert miles per hour into feet per second. The acceleration value of A=24 (feet per second per second) is probably enough to make the car's tires squeal. The driver accelerates more strongly when his speed differs most from the posted speed limit. The function deriv( ) has been provided with the independent variable, t, even though it does not use it. In more complicated problems it could be used: for example, our driver could brake to a halt after a minute, by inserting the following:

```
if (t > 60) vdesired=0;
```

The function called deriv( ) embodies all the information about the equations and the problem. Now we must solve it. Differential equations are usually solved by taking small steps in the independent variable. We can write a general purpose differential equation solver and pass it a pointer to the function that calculates the derivatives. Here is the simplest possible solver:

```
stepsolve(y,t,dt,ws,n,fun)
double y[];        /* dependent variables */
double t;          /* independent variable */
double dt;         /* step size for t */
double ws[]; /* working storage: n double words */
int n;             /* number of equations */
int (*fun)();      /* pointer to function */
{int i;

 (*fun)(ws,y,t);   /* calculate derivatives */
 for (i=0; i<n; ++i) y[i]+=dt*ws[i];
}
```

The solver is called with values for the dependent variables at time t. The function must be provided with a pointer to some working storage. After the function returns, the dependent variables are updated with values for time t+dt. The function

stepsolve( ) is called repeatedly in order to obtain a complete numerical solution to our system of differential equations.

```
#include "stdio.h"

main()
{double y[2],ws[2],t,dt;
 int deriv();
 int i;

 t=y[0]=y[1]=0.0;        /* initial conditions */
 dt=0.1;

 for (i=0; i<300; ++i)
        {stepsolve(y,t,dt,ws,2,deriv); t+=dt;
         if ((i%10) == 9)
              printf("t=%6.1f X=%8.3f V=%8.3f\n",
                     t,y[0],y[1]/FPS);
        }
}
```

If you were to run the program, you would see that the solution has an error of a few percent compared to an analytical solution. The accuracy of the solution could be doubled by halving the time step, but the length of time required to obtain a solution would double.

The errors are introduced in the function stepsolve( ). When the dependent variables are advanced from one time value to the next, the derivatives are assumed to be constant for the duration of the step. Since this is not generally true, the solution will not be exact. Numerous methods have been devised that are more accurate than stepsolve. If you are interested in the theory behind them, you may wish to browse through a book on numerical analysis, but we will present a method that has served us well over the years. It is called the Runge Kutta Merson (RKM) method after the three mathematicians who contributed to its development.

A function for performing a single step with the RKM algorithm is given below.

```
rkmstep(y,t,dt,ws,n,fun,e)
double y[];     /* dependent variables */
double t;       /* independent variable */
double dt;      /* step size for t */
double ws[];    /* working storage: 4n double words */
int n;          /* number of equations */
int (*fun)();   /* pointer to function */
double e[];     /* error estimate returned */
{double *dy,*a,*da,*b,*db,*c,*dc,*dy1;
 int i;

 dy=e;
 a=ws;
 da=ws+n;
 b=ws;
 db=ws+2*n;
 c=ws;
 dc=ws+n;
 dy1=ws+3*n;

 (*fun)(dy,y,t);
 for (i=0; i<n; ++i)
    a[i]=y[i]+dt*dy[i]/3.0;

 (*fun)(da,a,t+dt/3.0);
 for (i=0; i<n; ++i)
    b[i]=y[i]+dt*(dy[i]+da[i])/6.0;

 (*fun)(db,b,t+dt/3.0);
 for (i=0; i<n; ++i)
   c[i]=y[i]+dt*(0.125*dy[i]+0.375*db[i]);

 (*fun)(dc,c,t+0.5*dt);
 for (i=0; i<n; ++i)
   y[i]=y[i]+dt*(0.5*dy[i]-1.5*db[i]+2.0*dc[i]);

 (*fun)(dy1,y,t+dt);
 for (i=0; i<n; ++i)
   e[i]=dt*(2.0*dy[i]+8.0*dc[i]-9.0*db[i]dy1[i])/3.0;
}
```

This function differs from stepsolve( ) in several ways. It requires more working storage, and it also returns a set of extra values in the array e which is an estimate of the error of the solution. The function is dramatically more accurate than stepsolve( ). Halving the step size often reduces the error by a factor of thirty.

Because the RKM method estimates its own error, we can write an adaptive version that modifies its step size in order to maintain a prescribed accuracy. Here is such a function.

```
arkmstep(y,t0,dt0,ws,n,fun,e,nt)
double y[];          /* dependent variables */
double t0;           /* independent variable */
double dt0;          /* step size for t */
double ws[];/* working storage: 7n double words */
int n;               /* number of equations */
int (*fun)();        /* pointer to function */
double e[];          /* maximum error tolerated */
int *nt;             /* pointer to subdivision */
{int i,j;
 double *y0,*toterr,*e2,dt,t,err,ratio;

 y0=ws+4*n;
 toterr=ws+5*n;
 e2=ws+6*n;
 if (*nt < 1) *nt=1;

 for (j=0; j<n; ++j) y0[j]=y[j];

 tryagain:
 for (j=0; j<n; ++j) toterr[j]=0.0;
 dt=dt0/(*nt);
```

```
 t=t0;

 /* step out solution with *nt steps */
 for (i=0; i<(*nt); ++i)
        {rkmstep(y,t,dt,ws,n,fun,e2);
         t=t+dt;
         for (j=0; j<n; ++j)
              {err=e2[j]; if (err < 0.0) err=-err;
               toterr[j]+=err;
              }
        }

 /* is it good enough? */
 ratio=0.0;
 for (j=0; j<n; ++j)
        if (e[j] > 0.0)
                {err=toterr[j]/e[j];
                 if (err > ratio) ratio=err;
                }
 if (*nt < 128 && ratio > 1.0)
        {*nt*=2;
         for (j=0; j<n; ++j) y[j]=y0[j];
         goto tryagain;
        }

 /* is it too good? */
 if (*nt > 1 && ratio < 0.01) *nt/=2;
 return ratio < 1.0;
}
```

To use this function, you must provide it with yet more working storage and provide it with an upper limit on the size of the error that you can tolerate. In addition, you must pass it a pointer to an integer that is your guess of how many times the interval is to be subdivided. If your guess is too small, so that your accuracy criterion is not satisfied, the function will double the number of subdivisions and recalculate the solution. On the other hand, if the solution seems to be excessively accurate, the number of subdivisions is halved, so that you will have a better guess the next time you use the function.

Sometimes, perhaps because of rounding errors or an unreasonably small error tolerance, the function may not be able to satisfy the criterion. In that case there is a built in limit of 128 subdivisions, and the function returns a zero to tell you that the solution may be inaccurate. Usually the function returns one to let you know that the solution is accurate.

If you use the adaptive RKM method for our motorist problem, you will need to call the equation solver in the following way:

```
main()
{double y[2],ws[14],t,dt,e[2];
 int deriv();
 int i;
 int nt;

 t=y[0]=y[1]=0.0;        /* initial conditions */
 dt=1;
 nt=2;
 e[0]=e[1]=0.01;

 for (i=0; i<30; ++i)
        {
         arkmstep(y,t,dt,ws,2,deriv,e,&nt);
         t+=dt;
         printf("t=%6.1f X=%8.3f V=%8.3f nt=%d\n",
                t,y[0],y[1]/FPS,nt);
        }
}
```

You will notice that something strange happens when the car leaves the 30 MPH zone. Because the acceleration suddenly increases, a graph of velocity against time would show a kink. An equation solver like arkmstep( ) is designed on the assumption that everything changes smoothly, and it reacts to the kink by decreasing its step size to the minimum allowed in an attempt to maintain accuracy.

Dynamics

Now that we are able to solve differential equations, we are in a position to look at the dynamics of how objects move. If we restrict our attention to a single small object (mathematicians would call it a particle) of fixed mass M, then the equations describing its motion in one dimension are:

$$X' = V \text{ and } V' = F/M$$

where F is the force on the particle. We have neglected the effects of quantum mechanics which apply to very small objects and the effects of relativity which apply to very rapidly moving objects.

These equations imply that in the absence of any forces, an object once moving would keep on moving forever. Since everyday experience seems to be in conflict with this, it means that our environment must include forces that bring most things to rest quite quickly. When you set out to calculate the motion of an object, you must come up with a formula for each force that you wish to include.

Near the surface of our planet, the force of gravity takes a particularly simple form. The force is g*M, where g is the gravitational acceleration, approximately 32 feet per second per second. As we know, the force is directed downwards.

Frictional forces can take many forms. The frictional force of one object sliding on a rough surface is mu*g*M, where mu is called the coefficient of friction and is a number that is usually less than unity. This frictional force is directed in the opposite direction to the motion of an object and vanishes abruptly when the object comes to rest. This discontinuous behavior is what makes a car lurch after braking to a halt; it also displeases equation solvers like arkmstep( ), which abhor discontinuities.

The frictional force of air resistance depends upon the velocity of an object. The direction of the force is opposite to the velocity, so you could use the formula -C*V. The constant C depends upon the size and shape of an object, but not upon its mass. This is why a feather falls more slowly than a lump of lead. You can experiment with other forms of friction by raising the velocity to a power, say p. Ensure that the direction of the force is opposite to the velocity, or your object rapidly depart from the known universe. A value of p between one and two will produce quite realistic results.

When objects collide, impulsive forces come into play. When a falling tennis ball hits the ground, a brief force is applied that starts it bouncing back into the air. We can treat this force in two ways. An idealized force is infinitely powerful but is applied for an infinitely short time. Suppose we are using an equation solver to calculate the motion of a particle with position X and velocity V. If the ground is described by values of X that are less than zero, then we perform the following test after each use of argkstep( ):

```
if (X < 0) { X=-X; V=-e*V; }
```

Do not put this code in the derivative function, place it in the loop that calls the equation solver. The value e is called the coefficient of restitution and it must not be greater than unity or each bounce would be higher than the last.

The other way of treating impulsive forces is to include them in the derivative function. The force could be calculated as follows.

```
if (X < 0)
     if (V < 0)
          F=-A*X;
     else
          F=A*e*X;
else
     F=0;
```

The constant A determines the stiffness of the ground. If the ground is too stiff, the equation solver will have problems; if A is too small the object will penetrate the ground deeply before popping back.

When you design a dynamic animation, you will probably find that you have to tinker with the initial conditions and the parameters that describe the forces until everything works just the way you want it. Do not feel that you have to mimic nature exactly; exaggerate for creative effect.

Once you have designed your forces and have chosen an initial position and velocity for the particle, you can use an equation solver like arkmstep( ) to calculate its motion at any later time. Going from one dimension to three presents no problem: the pair of equations is replaced by six, two for each direction. If you know that all motion will be confined to a plane, then you can restrict your solution to two dimensions, and you only have to solve four equations.

As an example, let us calculate the motion of a particle fired upward from ground level at a 45 degree angle, subject to air resistance, and follow its motion through several bounces against the ground.

The derivative function is fairly simple.

```
#define g 32.0     /* gravitational acceleration */
#define M 1.0      /* mass of object */
#define C 1.0      /* air resistance coefficient */
#define A 8000.0   /* stiffness of the ground */
#define e 0.8      /* coefficient of restitution */

traj(dy,y,t)
double dy[4],y[4],t;
{double F0,F1;

 dy[0]=y[2];     /* X' = V  horizontally */
 dy[1]=y[3];     /* X' = V  vertically */

 F0=-C*y[2];     /* horizontal air resistance */
 F1=-C*y[3];     /* vertical air resistance */

 if (y[1] < 0)  /* impact with the ground */
        {if (y[3] < 0) F1=F1-A*y[1];
         else F1=F1-A*e*y[1];
        }

 F1=F1-g*M;      /* gravity */
 dy[2]=F0/M;     /* V' = F/M horizontally */
 dy[3]=F1/M;     /* V' = F/M vertically */
}
```

In order to be able to use our trajectory, the data needs to be converted into a form that is compatible with your three dimensional modelling program. We provided some functions in Volume 1 issue 5 that generate Sculpt 3D compatible scene files. Here is a function that solves the differential equations and produces a scene file.

```
main()
{double y[4],ws[28],t,dt,e[4];
 int traj();
 int i;
 int nt;
 struct object object;
 struct vertex *v0,*v1;

 t=0.0;            /* initial conditions */
 y[0]=0.0;             /* horizontal coordinate */
 y[1]=0.0;             /* vertical coordinate */
 y[2]=20.0;            /* horizontal velocity */
 y[3]=20.0;            /* vertical velocity */
 dt=1.0/30.0;          /* two jiffies */
 e[0]=e[1]=e[2]=e[3]=0.01;  /* tolerate error */
 nt=2;

 object.vertices=0;
```

```
    object.edges=0;
    object.faces=0;
    v0=object.vertices =
      newvertex(y[0],0.0,y[1],object.vertices);

    for (i=0; i<120; ++i)  /* four seconds worth */
       {arkmstep(y,t,dt,ws,4,traj,e,&nt); t+=dt;
        printf("t=%6.1f X=(%8.3f,%8.3f)\
               V=(%8.3f,%8.3f) nt=%d\n",
               t,y[0],y[1],y[2],y[3],nt);
        v1=object.vertices =
           newvertex(y[0],0.0,y[1],object.vertices);
        link(&object,v0,v1,0L);
        v0=v1;
       }
    wrscene("path.scene",&object);
  }
```

The illustration shows the trajectory with realistic looking bounces. The ground stiffness is set so that the particle only penetrates the ground a small amount.

Next time

The next article will apply this simple treatment of dynamics so that you can animate any three dimensional solid object and include the effects of rotation and elasticity.

. . . if you have at least 3 years of assembly language programming experience, preferably one year of 68000 or have an NTSC/PAL Video Hardware background. Join the team that makes no compromises and help us change the world. Key positions available at the fastest growing Amiga company. Hackers call Stephanie at 913-354-1146 or write to Stephanie at 115 West Crane Street, Topeka KS 66603.

☆ ☆ ☆ **ReSource** ☆ ☆ ☆

Intelligent Interactive Disassembler for Amiga Programmers

"**ReSource** *makes full use of the Amiga windowing environment . . . One of the features that distinguish* **ReSource** *is its ability to produce output which is compatible with most popular assemblers . . . a bargain in this world of overpriced software." -- Andre Marino*

ReSource is an interactive disassembler for the Amiga computer. Entirely hand-coded in assembly language, this is the most powerful disassembler available on ANY personal computer! Over 700 menu functions allow you to:

- Load from ANY file, read tracks, or disassemble memory directly.
- Save ALL, allows you to continue disassembling later.
- Save to a ".asm" file, partial saves supported.
- Automated symbol creation:
  JSR -$1E(A6) becomes JSR _LVOOpen(A6)
  MOVE.L #$3EE,DO becomes MOVE.L #MODE_NEWFILE,DO
  Over 5,000 symbols supported!
- Incredibly smooth scrolling.
- Full key rebinding supported. Load/save keytables.
- Up to 38 user-defined macros available. Load/save/overlay macros, looping/conditionals supported, variable execution speed.
- Numeric and text processing functions allow you to automate many things that used to make disassembling so slow!
- Search function uses full ARP wildcards.
  Search forwards, backwards, or use the special bi-directional search!
- The fastest text rendering available on the Amiga.
- Forwards and backwards referencing allows you to quickly examine referenced code/data, and return to your original position.
  Nested references supported.
- Automatically/manually create labels. Manually create/edit/delete labels, symbols, end-of-line comments, full-line comments.
- Special support for "C" programs, including "A4" register data referencing.

If you're serious about disassembling code, look no further!

To order ReSource V3.01 contact :

Send Check or Money Order. No COD.
Personal checks will be held for 10 days.
(Credit card facilities available soon)

The Puzzle Factory
P.O. Box 986
Veneta, OR 97487 USA
(503) 935-3709

Glen McDiarmid
28 Marginson St.
Ipswich, QLD
4305 AUSTRALIA
(07) 812-2963

First Steps With C++

Object-oriented programming on the Amiga

by Richie Bielak

Richie Bielak has been programming for 10 years, mostly in Modula-2 on VAXen. He currently works as a programmer in telecommunications for a bank. His leisure hours are spent either with his four year-old Amiga 1000 or flying an airplane.

C++ is a recent addition to the family of programming languages available on the Amiga. C++ was developed at Bell Labs in the early 1980s; its port to the Amiga has been fairly quick. The main reason for this is that C++ can be implemented via a preprocessor that translates C++ to C.

On the Amiga, a C++ system is being sold by Lattice as an addition to its C compiler. The package contains a port of the AT&T C++ preprocessor, along with the Lattice 4.0 C compiler and linker. Although the compiler is somewhat slow, Lattice C++ does provide a nice tool for programming the Amiga.

C++ and object-oriented languages

C++ is a hybrid language. On one hand, it is an extension of C; on the other, it contains a number of features taken from *object-oriented* languages. Everyone is familiar with C, but what is object-oriented programming?

Simply put, object-oriented programming is just a different way of thinking about writing programs. The usual buzz-word for 'a way of thinking' is a *paradigm*. In the old programming paradigm, the code and the data were considered as separate entities. To write a program, one designed a data structure, then devised an algorithm and finally wrote the code. A popular book by Niklaus Wirth describing this approach to programming was called *Algorithms + Data Structures = Programs*.

In the object-oriented approach, related data and code are tied together into an *object*. A program is constructed by defining objects that describe the problem, and the interaction between them consitutes the computation. On the surface the object-oriented approach does not seem that different from the conventional programming paradigm, but it does have a number of advantages. For example, object-oriented programming encourages hiding of implementation details, thereby making the resulting programs easier to read, write and change.

Object-oriented programming forces us to think and program in a new way. As a friend of mine used to say, "If all you've got is a hammer, then everything looks like a nail". Maybe with object-oriented languages we can notice other objects (pun intended) among the nails!

Classes in C++

The most important feature introduced in C++ is the concept of a *class*. A class describes a type of an object, in a manner analogous to a C structure. However, a class not only specifies what kind of data will be part of the object, it also defines the associated functions. Functions declared as a part of a class are called *member* functions of the class.

The data and functions in a class can be divided into *public* and *private* sections. As the name indicates, public data and functions can be accessed by any user of the object. On the other hand, private data and functions can only be used by the member functions. To see how this works, let's consider an example of a class that defines a data structure we'll call *stack*:

```
class stack {
   private:
      int *stackp;
      int max, top;
   public:
      // Public member functions
      void push (int item);
      int  pop ();
      // Constructor and destructor functions
      stack (int max_entries);
      ~stack () {delete stackp; top = 0; max = 0;}
   }; // stack <- this is a comment
```

The class *stack* has three private data items. They are: *stackp*, the pointer to the area of memory allocated for the stack; *max*, the maximum number of items that can be placed on the stack; and *top*, the index of the stack's top.

The public part has only member functions. Two of those, push(int item) and pop( ), perform the stack operations. The functions stack(int max_entries) and ~stack( ) play a special role.

Function stack(int max_entries) is the *constructor* function for the class *stack*. The constructor function is invoked automatically whenever an object of the class is declared. As we will see, this function will perform initialization of the private variables. The function ~stack( ) (yes, its name starts with the tilde (~) character) is referred to as the *destructor* function. It is called when the object is deleted.

Here is an example of how the class *stack* might be used in a program:

```
void test_stack () {
   stack s1(10);  // set up a stack of 10 elements

   s1.push (1);    // call a member function
   s1.push (2);
   // when the function exits,
   //the destructor is called to delete s1
}
```

When the variable *s1* is declared, the constructor function is called. The number 10 is a parameter for the constructor, which specifies that a stack for 10 integers is to be created. The above code also illustrates how the member functions are called. Note that *s1* is a local variable created upon entry to the function, and destroyed when the function exits. This means that the destructor for *s1* will be called when the function test_stack( ) exits.

Now let's look at the implementations of the member functions. First, note that the destructor function is declared together with its code in the class definition, like this:

```
~stack () {delete stackp; top = 0; max = 0;}
```

Such a declaration is called *inline*. The C++ compiler will insert the function's code directly into the compiled program, thereby eliminating function call overhead. Considering that most data items in C++ will be accessed via member functions, this optimization makes a lot of sense.

The other member functions are implemented below:

```
// Constructor
stack::stack (int max_entries) {
   stackp = new int[max_entries-1];
   max = max_entries-1; top = 0;
}
// Push an item
void stack::push (int item) {
   if (top < max) stackp[top++] = item;
   else
      printf ("Error - Stack overflow..\n");
}
// Pop an item
int stack::pop() {
   if (top > 0) return(stackp[--top]);
   else
      printf ("Error - Stack underflow...\n");
}
```

Notice that the name of each function is preceeded by the name of their class and two colons. That's how we can tell that these functions belong to the class *stack*.

Careful reader will also notice other differences between C++ and C – the use of function prototypes, for example. Also note that the functions *new* and *delete* are used to allocate and deallocate memory.

Derived classes

Given an existing class, it is possible to create a new class called the *derived* class. The derived class contains all the data and functions of its *parent* class, but it also can contain new data and new functions.

We will create a new class called *e_stack* (for extended stack) that will be derived from *stack*. It will add another member function. Here is the declaration of *e_stack*:

```
class e_stack : public stack {
   public:
      int examine_top ();
};
```

Note the name of the parent class, *stack*, occuring after the colon in the first line. Also note the keyword *public*, which is here to make all member functions of *stack* accessible to users of *e_stack* – we still want to apply push(int item) and pop( ) to *e_stack* objects. If this keyword were missing, the member functions of the parent class could not be applied to the derived object.

Our derived class adds another function, called examine_top( ). This function returns the value of the top item on the stack. Here is its implementation:

```
int e_stack::examine_top() {
   int it = pop ();
   push(it);
   return (it);
}
```

Notice that we have to use the functions pop( ) and push(int item) to remove and put back the top element. This is because we cannot access the private data of *stack*, even from the derived class.

In most cases accessing private data via member functions is the best way to go. However, in some derived classes it would be useful to read and write the private data directly, perhaps for the sake of efficiency. This arrangement is possible, through *friend* classes.

To allow an object of class *e_stack* access to private data of *stack*, we would have to declare e_stack as a friend class in the declaration of *stack*. The begining of *stack*'s declaration would be changed thus:

```
class stack {
   friend class e_stack;
   .... rest of the declaration....
```

The corresponding implementation of examine_top( ) would be like this:

```
int e_stack::examine_top () {
   return (stackp[top-1]);
}
```

As you can see, friend classes can be useful. However, one must be careful not to abuse the friend class construct. Creating too much interdependence between separate classes will result in programs that are hard to modify.

Intuition and C++

Now that we know what classes are, let's see how they can be applied in programming of the Amiga. We begin by considering a class that simplifies creation and deletion of Intuition windows.

To open a window, a program must first set up a *NewWindow* structure and call the function OpenWindow( ). Since the *NewWindow* structure contains a lot of fields, it is a nuisance to set up. We will create a more convenient way of doing the same thing.

In C++ we can define a class called *simple_window*, which will make creating windows easier. You can see the declaration of the *simple_window* class in Listing 1. By the way, this declaration would normally be kept in a file called *simple_window.h*, so that it can be included by all users of the *simple_window* class.

The public portion of *simple_window* contains three member functions. They are the constructor, the destructor, and the function get_msgw( ). The constructor function opens the window and the destructor closes it. The function get_msgw( ) is used to remove messages from the window's IDCMP port. It differs from the usual GetMsg( ) function, in that get_msgw( ) will wait for a message to arrive before returning control to the caller.

Using the *simple_window* class is easy. To open a window, just declare an object of the class. Here is an example:

```
simple_window test_window (0, 0, 400,
   100, "Test window");
```

The above line will open a window positioned in the upper left hand corner, whose width is 400 pixels and height 100 pixels. Now let's compare the above code, with the declaration of a *simple_window* constructor function below:

```
// Constructor
simple_window (const USHORT left, const USHORT top,
   const USHORT width, const USHORT height,
   const TEXT *title,
   const LONGBITS flags = NULL,
   const IDCMPFlags idc = NULL,
   const PGadget gp = NULL);
```

Notice that we did not pass any values for the last three parameters: *flags*, *idc* and *gp*. By default, these are set to NULL. The actual code for the *simple_window* constructor function sets the window flags and IDCMP flags to certain default values when *flags* and *idc* are omitted.

Listing 2 contains the implementation of the simple_window( ) and get_msgw( ) functions. The code for simple_window( ) is straightforward and its examination is left to the reader. To see more C++ features, let's take a look at get_msgw( ):

```
IntuiMessage *simple_window::get_msgw() {
   IntuiMessage *mp = wp->message();
   if (mp == NULL) {
      mp = (IntuiMessage *)wp->UserPort->wait();
      mp = wp->message();
   }
   return mp;
} // get_msgw
```

In the above code we use the object *wp* from the private portion of the class. *Wp* is a pointer to class *Window* (the class definition for *Window* is provided with the Lattice C++ compiler). Class *Window* contains the data Intuition keeps for an open window. In addition, *Window* has a member function called message( ). This function returns the first message from the window's IDCMP port, or NULL if no message exists.
The line

```
IntuiMessage *mp = wp->message();
```

declares a pointer to an *IntuiMessage* structure and assigns its value by calling the function message( ). Since *wp* is a pointer to the *Window* class, it is used to access the member( ) function; hence the expression `wp->message()`.

A similar situation occurs when the member function wait( ), from class *MsgPort* is called. Here we have to use a double de-reference, since a pointer to the port is a field in the *Window* class; hence the expression `wp->UserPort->wait()`.

Deriving window objects

As it stands, the *simple_window* class is not that useful. However, it can serve as a base for deriving more useful classes. One such derived class is the *text_window* class, also declared in Listing 1.

The *text_window* class adds two member functions to *simple_window*. They are put_line(char *s) and clear( ). The put_line(char *s) function is used to write a line of text into the window, and clear( ) erases all written text.

The declaration of *text_window* class contains two data items in the private section: the pointer to the *RastPort* needed for writing text, and a variable to remember where the next line is to be written. Since the private variables must be initialized, the class *text_window* requires a constructor function. On the other hand, *text_window* does not need a destructor function. The one from its parent is sufficient to close the window.

Now, *text_window* class is derived from *simple_window*. Since both classes possess constructor functions, which constructor is invoked first? The rule is that the parent's constructor is called first. Therefore, in the constructor for *text_window*, we only need to initialize *text_window*'s private data.

The code for *text_window* constructor is shown below (also see Listing 2):

```
text_window::text_window (const USHORT left,
      const USHORT top,
      const USHORT width, const USHORT height,
      const TEXT *title,
      const LONGBITS flags = NULL,
      const IDCMPFlags idc = NULL,
      const PGadget gp = NULL)
   :
      (left, top, width, height, title, flags, idc, gp)
   {
   cur_row = 18;
   rp = wp->graphic();
   rp->move (5, cur_row);
}
```

First, note that the parameter list for the text_window constructor function is followed (after the single colon) by a list of the parameters that will be passed to the parent's constructor. In this case, all parameters are passed.

Second, since *text_window* is declared as a friend of *simple_window*, it is able to use the variable *wp* from the private section of *simple_window*.

An Intuition gadget class

To implement our example program we need another class that will allow us to set up gadgets. Listing 3 contains the definition of class *button_gadget*. A button gadget is a boolean gadget that the user can click on. The gadget is drawn as a box around piece of text. Note that *button_gadget* class is derived from class *Gadget*, which is also supplied with Lattice C++. It consists entirely of public data that is part of Intuition's *Gadget* structure.

The only parameters required for *button_gadget*'s constructor are the position of the gadget in the window and the text that will appear inside the box. For example, here is how a *button_gadget* object may be declared:

```
button_gadget ok (10, 10, " OK ");
```

In addition, *button_gadget* function takes one optional parameter: a pointer to the next gadget in a list. This parameter will be required when more than one gadget is being attached to a window.

The *button_gadget* class has no destructor function – no memory is being allocated in gadget creation, so no cleanup is required.

There is one more member function declared in the public part of this class. This function, called id( ), returns the *GadgetID* field of the *Gadget* class. Id( ) is declared inline, and it looks like this:

```
USHORT id () { return (this->GadgetID); };
```

The word *this* used above is a C++ keyword. It is an implicit parameter to any member function, and it is a pointer to the object that the function is currently working on.

The private section of *button_gadget* class contains data needed for the setup of the gadget. Among the variables are *Border* and *IntuiText* structures. In addition, there are two private member functions that make writing the constructor a little easier.

The complete code for the constructor is found in Listing 4. Again, note the use of the *this* keyword.

A C++ file reader

We can now put all of our work together into a complete program. This program will take a file and display it in a window. To make things simple, the program will read from standard input; to display a file, then, one has to use redirection operators.

The file will be displayed one page at a time. At the bottom of the window we will have two button gadgets, **Cancel** and **Next**. If the **Next** gadget is hit, the next page is shown. If either the **Cancel** or window close gadget is selected, the program exits.

The code for the program is found in Listing 5. To see how the work we have done with the classes above pays off, consider the code needed to open a window with two gadgets (this code is found in main( )):

```
button_gadget cancel (20, 180, "Cancel");
button_gadget next   (540, 180, " Next ",
      (Gadget *)&cancel);
text_window win   (0, 0, 640, 200, "More...",
      WINDOWCLOSE | ACTIVATE | WINDOWDRAG |
      WINDOWDEPTH, CLOSEWINDOW | GADGETUP,
      (const Gadget *)&next);
```

The remainder of the code is straightforward; the reader is encouraged to looked at it carefully.

Listing 1: Class definitions

```
// ++++++++++++++++++++++++++++++++++++++++++++++
//
//   FILE: "simple_window.h"
//
//    This file contains class definitions for classes that
//    simplify the use of Intuition windows
//
//    Created: 2/9/89 by Richie Bielak
//
// ++++++++++++++++++++++++++++++++++++++++++++++

#include <exec/types.h>
#include <intuition/intuition.h>

// +++++++++++++++++++ simple_window ++++++++++++++
class simple_window
{
  friend class text_window;

  private:
    Window *wp;

  public:

    // Constructor
    simple_window (const USHORT left, const USHORT top,
            const USHORT width, const USHORT height,
        const TEXT *title,
        const LONGBITS flags = NULL,
        const IDCMPFlags idc = NULL,
        const PGadget gp = NULL);

    // Destructor
    ~simple_window () { delete wp;};

    // Get next message from Intuition; wait for one to arrive
    IntuiMessage *get_msgw();

    // Function to check if window opened ok.
    BOOL window_is_open () { return (wp != NULL);};
};

// +++++++++++++++++++ text_window ++++++++++++++++
//
class text_window : public simple_window
{
  private:
    USHORT cur_row;
    RastPort *rp;

  public:

    text_window ( const USHORT left, const USHORT top,
              const USHORT width, const USHORT height,
          const TEXT *title,
          const LONGBITS flags=0, const IDCMPFlags idc=0,
          const PGadget gp = NULL);

    void put_line (char *s);
    void clear ();

};
```

Listing 2: Window object method implementations

```
// ++++++++++++++++++++++++++++++++++++++++++++++
//
//   FILE :"simple_window.cp"
//
//    This file contains implementation for classes that
//    simplify the use of some of the Intuition's objects
//
//    Created: 2/9/89 by Richie Bielak
//
// ++++++++++++++++++++++++++++++++++++++++++++++

#include <exec/types.h>
#include <intuition/intuition.h>
#include "simple_window.h"


// ++++++++++++++++++++++++++++
simple_window::simple_window
              (const USHORT left, const USHORT top,
             const USHORT width, const USHORT height,
          const TEXT *title,
          const LONGBITS flags = NULL,
              const IDCMPFlags idc = NULL,
              const PGadget gp = NULL)

{
   const LONGBITS   default_flags = WINDOWCLOSE | WINDOWDRAG | ACTIVATE;
   const IDCMPFlags default_idcmp = CLOSEWINDOW;

   NewWindow nw;

   nw.LeftEdge = left;       nw.TopEdge  = top;
   nw.Width    = width;      nw.Height   = height;
   nw.DetailPen = 2;         nw.BlockPen  = 1;
   nw.FirstGadget = gp;      nw.CheckMark = NULL;
   nw.Title = title;         nw.Screen = NULL;
   nw.BitMap = NULL;         nw.Type = WBENCHSCREEN;
   nw.MinWidth = 10;         nw.MinHeight = 10;
   nw.MaxWidth = 640;        nw.MaxHeight = 400;

   nw.IDCMPFlags = (idc == 0) ? default_idcmp : idc;
   nw.Flags      = (flags == 0) ? default_flags : flags;

   // Create a window
   wp = new Window (&nw);

};

IntuiMessage *simple_window::get_msgw()
{
  IntuiMessage *mp = wp->message();
  if (mp == NULL) {
    mp = (IntuiMessage *)wp->UserPort->wait();
    mp = wp->message();
  }
  return mp;
};

// +++++++++ routines for text_window ++++++

text_window::text_window
               (const USHORT left, const USHORT top,
              const USHORT width, const USHORT height,
          const TEXT *title,
          const LONGBITS flags = NULL, const IDCMPFlags idc = NULL,
          const PGadget gp = NULL)
      :
               (left, top, width, height, title, flags, idc, gp)
  {
    cur_row = 18;
    rp = wp->graphic();
    rp->move (5, cur_row);
  };

void text_window::put_line (char *s)
 {
   *rp << s;
   cur_row += 8;
   rp->move (5, cur_row);
 };
```

```
void text_window::clear()
 {
    // Save value of current pen.
    int pen = rp->FgPen;

    SetAPen (rp, 0);
    RectFill (rp, wp->LeftEdge+3, wp->TopEdge+10,
                  wp->Width-3, cur_row);

    // Restore pen value
    SetAPen (rp, pen);
    cur_row = 18;
    rp->move (5, cur_row);
 };
```

Listing 3: A derived class definition

```
//
//    FILE: "simple_gadget.h"
//

#include <exec/types.h>
#include <intuition/intuition.h>

// +++++++++++++++++ text_gadget ++++++++++++++
class button_gadget : Gadget
{
  private:
    SHORT pairs1[10];
    Border bd1;
    SHORT pairs2[6];
    Border bd2;
    IntuiText it;

    Border *setup_border ();
    IntuiText *setup_itext (const char *text);

  public:

    // contructor
    button_gadget (const SHORT left, const SHORT top,
                   const char *text, Gadget *next = NULL);

    USHORT id () { return (this->GadgetID); };

};
```

Listing 4: Method implentations for *button_gadget* objects

```
//
//  FILE: "simple_gadget.cp"
//

#include "simple_gadget.h"

//
IntuiText *button_gadget::setup_itext (const char *text)
{
   it.FrontPen = 1;     it.BackPen  = 0;
   it.DrawMode = JAM1;
   it.LeftEdge = 5;     it.TopEdge = 2;
   it.ITextFont = NULL; it.IText = (char *)text;
   it.NextText = NULL;
   return &it;
};

//
Border *button_gadget::setup_border ()
{
   // Border #1
   bd1.LeftEdge = 0;    bd1.TopEdge = 0;
   bd1.FrontPen = 1;    bd1.BackPen = 0;
   bd1.DrawMode = JAM1; bd1.NextBorder = &bd2;
   bd1.Count = 5;    bd1.XY = &pairs1[0];
   pairs1[0] = 0;          pairs1[1] = 0;
   pairs1[2] = this->Width; pairs1[3] = 0;
   pairs1[4] = this->Width; pairs1[5] = this->Height;
   pairs1[6] = 0;          pairs1[7] = this->Height;
   pairs1[8] = 0;          pairs1[9] = 0;

   // Border #2
   bd2.LeftEdge = 0;    bd2.TopEdge = 0;
   bd2.FrontPen = 2;    bd2.BackPen = 0;
   bd2.DrawMode = JAM1; bd2.NextBorder = NULL;
   bd2.Count = 3;    bd2.XY = &pairs2[0];
   pairs2[0] = this->Width+2; pairs2[1] = 2;
   pairs2[2] = this->Width+2; pairs2[3] = this->Height+1;
   pairs2[4] = 2;             pairs2[5] = this->Height+1;

   return &bd1;
};

//
button_gadget :: button_gadget (const SHORT left, const SHORT top,
                                const char *text, Gadget *next)
{
   static USHORT id = 0; // Used to assign unique ID to every gadget

   this->NextGadget = next;
   this->LeftEdge   = left; this->TopEdge  = top;

   // Width and height must be based on size of text
   this->Width    = strlen (text) * 9;
   this->Height = 11;
   this->SelectRender = NULL;     this->MutualExclude = NULL;
   this->GadgetRender = NULL;     this->SpecialInfo = NULL;
   this->Flags      = GADGHCOMP;
   this->Activation = RELVERIFY | GADGIMMEDIATE;
   this->GadgetType = BOOLGADGET;

   // Set the gadget ID.
   this->GadgetID = id; id++;
   this->UserData = NULL;
   //
   // Set up the "IntuiText" structure and border
   this->GadgetText   = setup_itext (text);
   this->GadgetRender = setup_border ();
};
```

Listing 5: A C++ file reader program

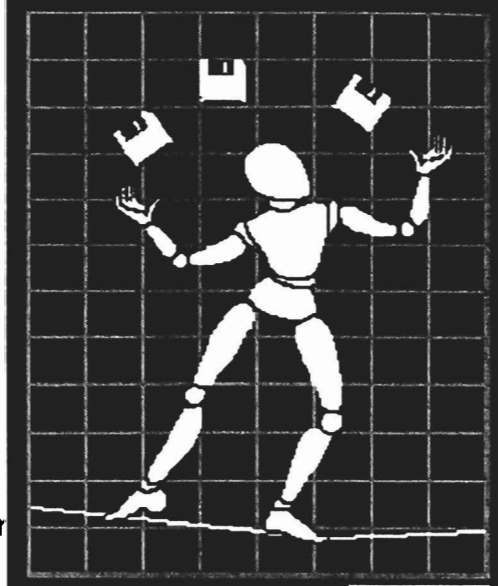
```
//
// An Amiga style "more" program.
//
// Created: 3/3/89 by Richie Bielak
//
//      FILE: "more.cp"
//
#include <stdio.h>
#include "cpp:simple/simple_window.h"
#include "cpp:simple/simple_gadget.h"

const int cancel_id = 0;
const int   next_id = 1;


// +++++++++++++++
BOOL get_line (char *s)
{
  const int EOL = 10;
  const int TAB = 9;
  int i = 0;

  do {
    char ch = getchar ();
    switch (ch) {
```

```
      case EOF: return FALSE;
      case EOL: s[i] = 0;
                return TRUE;
      case TAB: for (int j=1; j<=8; j++) {s[i++] = ' ';}
                break;
      default:  s[i++] = ch;
    } // switch
  } while (TRUE);
} // get_line

// ++++++++++++++++
BOOL show_page (text_window *wp)
{
  const lines_per_page = 17;
  int   lcount = 0;
  char  line [80];
  BOOL  done;

  wp->clear ();
  while ((lcount++ <= lines_per_page) &&
         (done = get_line (line))) {
    wp->put_line (line);
  } // while

  wp->put_line (""); // Skip one line
  if (done)
    {wp->put_line (" ---- more ---- ");}
  else
    {wp->put_line (" ---- EOF ----");}
  return (!done);
} // show_page

// ++++++++++++++++
int process_msg (text_window *wp)
{
   BOOL done = FALSE;
   IntuiMessage *mp = wp->get_msgw();

   switch (mp->Class) {
     case CLOSEWINDOW:
       done = TRUE;
       break;
     case GADGETUP:
       button_gadget *gp = (button_gadget *)mp->IAddress;
       done = (gp->id() == cancel_id);
       break;
   } // switch
   mp->reply();
   return (done);
} // process_msg

main()
{
   // Set up two gadgets and open the window
   //
   button_gadget cancel (20, 180, "Cancel");
   button_gadget next   (540, 180, " Next ", (Gadget *)&cancel);
   text_window win   (0, 0, 640, 200, "More...",
                     WINDOWCLOSE | ACTIVATE | WINDOWDRAG | WINDOWDEPTH,
                     CLOSEWINDOW | GADGETUP,
            (const Gadget *)&next);

   if (win.window_is_open ()) {
     BOOL finished = FALSE;
     BOOL canceled = FALSE;
     do {
       finished = show_page (&win);
       canceled = process_msg (&win);
     } while ((!finished) && (!canceled));
   } // if
   else
     printf ("Sorry, unable to open a window.\n");
}
```


NEW Release ver 1.10

- Full Intuition based Interface
- Selective Restore
- Amiga wildcards
- Multiple indexing for greater **safety**
- Over 907K per floppy

saf-T-net



Hard Disk Backup

for your Amiga

available at your dealers **now**

or from (714) 955-2555

Metadigm, Inc. 19762 MacArthur Blvd. Suite 300; Irvine CA 92715

SAFE HARBOR SOFTWARE & PERIPHERALS

Great Prices! Shipping based on weight and zone

For Information & Catalog Call
Voice 414-544-2066
Personal BBS 414-544-6567

Software

Atalk III	$ 64.69
ARexx	$ 31.10
Audio Master II	$ 59.10
Aztec C Developer	$195.00
Aztec C Professional	$130.00
B.A.D	$ 33.00
Battlehawks	$ 33.89
Broadcast Titler	$195.00
CLIMate	$ 27.73
CygnusEd	$ 64.69
Deluxe Paint III	$105.00
Devpac Amiga	$ 64.69
Digi View Gold	$143.10
DOS 2 DOS	$ 36.39
Enigma Device	$ 27.73
FACC II	$ 22.00
Falcon	$ 31.00
GOMF 3.0	$ 27.28
Hi Soft BASIC Professional	$115.00
Inovatools	$ 52.37
Interfont	$ 76.00
Lattice C 5.0	$190.00
Lords of the Rising Sun	$ 33.89
Microfiche Filer +	$111.00
Movie Setter	$ 60.00
PageStream	$131.00
Power Windows 2.5	$ 59.09
Professional Draw	$130.00
Project D	$ 32.00
Pro Sound Designer	$111.00
Quarterback	$ 46.21
SuperBack	$ 47.90
SuperBase Personal 2	$ 90.00
SuperBase Professional 3.0	$200.00
SuperPlan	$ 99.97
Turbo Silver	$120.00
Video Effects 3D	$128.00
Wordperfect Library	$ 94.00
WShell	$ 31.10

Abacus Books/Sony Diskettes

Amiga C for Beginners	$ 15.00
Amiga DOS Inside/Out	$ 16.30
Amiga Machine Language	$ 16.30
Amiga System Programmer	$ 26.25
DS/DD Sony 3.5" Disks (50)	$ 72.00
ROM Kernal Ref. Manual	$ 27.50

Hardware

68030 GVP 25MHz	Call
501 Memory Clones	$170.00
Amigen	$145.00
CA-880 3.5" Floppy Drive	$155.00
Electronic Color Splitter	$ 80.00
Floppy Drive 3.5" (AIR) ext.	$169.95
Floppy Internal 2000	$120.00
Future Sound	$135.00
Impact A2000 HC/45	$785.00
Live! 2000	$315.00
Midget Racer CSA	$350.00
Midi Gold	$ 67.00
Mini Megs 512K	$325.00
Modem, Supra 2400 Int.	$145.00
My-T-Mouse	$ 45.00
Spirit Brd 500/1000 0K	$212.56
Sub System 500	$235.00
Supra SCSI Controller 2000	$205.00

Orders Only Please:
Outside WI: **800-544-6599**
Inside WI: **414-544-6599**
Visa/MC/CODs

2414 Pendleton Place ■ Waukesha, WI 53188 ■ 9 AM to 5 PM M-F

A Guru Meditates On Interrupts

Book one in a new series for Amiga programmers

Review by Nick Sullivan

Guru's Guide To The Commodore Amiga
Meditation #1 - Interrupts
From Sassenrath Research, P.O. Box 1510, Ukiah, CA, 95482 USA. 105 pages, including index. Price: $14.95 US

Among the various parts of the Amiga's voluminous operating system, it is Exec, the kernel of the Kernel, whose reputation has tarnished least. You may hear that the Graphics library is slow, that Intuition is buggy, that Workbench is a hack, that AmigaDOS... well, everybody knows about AmigaDOS.

Ask about Exec, and you are likely to be told that it is concise, efficient, clean - the elegant nucleus around which the rest of the Amiga's internal software revolves.

That being so, it's perhaps not surprising that this book on Amiga interrupts by Exec's principal author, Carl Sassenrath, is also concise, efficient and clean. Anyone who has delved into the often sketchy and sometimes impenetrable descriptions of Exec topics in Addison-Wesley's ROM Kernel Manuals will be overjoyed on turning to this book: not only is Sassenrath the ultimate authority on what makes Exec tick, but he also knows how to document even an abstruse topic like interrupts in clear and comprehensible terms. In short, this is one of the finest pieces of technical writing I have read.

The first of the book's three chapters is an introduction - to the author, to the series of which this is the first instalment, and to the topic. It begins with a recollection (or a flashback) of the earliest, visionary days in Amiga history ("The country... no the world, must view and hear this bouncing ball..."), but settles quickly into the business at hand with a quick overview of Exec in general and of interrupts in particular.

The second chapter, 'Principles', wastes no time in establishing the importance of interrupts in the Amiga system:

"The Amiga is a busy machine. With its custom VLSI chip-set it has the capability of performing many operations at once. In addition to the activities of its Motorola 68000 microprocessor, the Amiga may simultaneously be displaying graphics images on the screen, playing music or sound effects on four separate audio channels, moving or manipulating blocks of data with the blitter, transferring data to or from floppy disk, sending or receiving data on its serial and parallel ports, synchronizing and controlling hardware with its coprocessor, receiving characters from its keyboard and mouse, and timing various intervals.

"Each of these activities occurs on demand, at its own speed, without any knowledge of the other activities that share the same moment in time. Yet, all of the activities must at some point bid for the attention of their controlling software executing on the 68000 microprocessor. They achieve this through the use of interrupts. Only with a robust interrupt system does this degree of parallelism become practical."

The hardware that monitors the many sources of interrupts is known as the AIC (Amiga Interrupt Controller). The AIC distinguishes 14 interrupt sources; it maps the signals it receives from these onto the three IPL lines of the 68000, allowing six priority levels (NMI is not used). Later on, Exec examines the AIC's memory-mapped INTENAR register to determine which device the interrupt request originated from. For each interrupt source, there is an interrupt *handler* routine invoked by Exec. Some handlers (like the one for the vertical blank interrupt) are mostly responsible for calling a series of independent subroutines known as interrupt *servers*, which may be added to or removed from the *server chain* for that handler by external tasks.

Within this broad framework there are, as you might imagine, myriad gory details, yet Sassenrath somehow manages to remain lucid throughout the whole exposition. Having dealt with hardware interrupts, he goes on to discuss the mechanics of software interrupts, and even devotes a page or two to the unhappily undersupported task interrupts (or 'task exceptions') - a good idea that never really got off the ground.

The third chapter is called 'Programming', and consists mostly of brief, to-the-point examples relating to topics introduced in chapter two. Most of the code is in C, with some short stretches of assembler. This chapter also covers the interrupt-related operating system calls (AddIntVector(), Cause(), etc.), in some cases providing information on their use that is lacking in the RKM.

A book on Amiga interrupts is obviously not destined for mass-market success, but *this* book should be of value to any serious Amiga programmer, if only because it will enrich his understanding of how the machine works at its lower levels.

A Sampler from AmigaTEX

A Demonstration of LATEX in TRANSACTOR FORMAT

by Dick Barnes

Copyright ©1989 Dick Barnes

Eric Giguère reviewed AmigaTEX in general in a previous *Transactor*.[1] Although the program I use here, LATEX, is furnished as part of AmigaTEX, you must buy and study *The LATEX User's Guide and Reference Manual*, published by Addison-Wesley, before you attempt to use the program.

Because the printed output of a sophisticated text-processor illustrates it better than bushels of words, this article is printed from LATEX at 1270 dpi, although I drafted it with a 24-pin printer. I formatted this page to match that of the *Transactor* as closely as possible, simply to demonstrate the flexibility of LATEX.

What is LATEX?

LATEX is a set of five, major "canned" formats, plus the commands to invoke and modify them. As with TEX, it runs only from CLI. Each of the unmodified formats is far easier to use than TEX because Leslie Lamport, the author, wrote the complex TEX macros for you—and because he provides simple ways to solve the formatting problems you normally encounter. Here's an example: Each item in the following list of formats is bulleted because the list begins and ends with the simple macro `\begin{itemize}...\end{itemize}`.

- Books, with chapters, tables of contents, bibliographies, and indices.
- Articles, optionally with the material above, plus abstracts and title pages (this article is in modified Article format).
- Reports, with similar options.
- Letters: You can send them to multiple addressees and can print mailing labels.
- Slides (these I haven't explored).

[1] Vol. 1, Issue 5, Feb. '89, p. 66. AmigaTEX is available from Radical Eye Software, Box 2081, Stanford, CA 94309. Tomas Rokicki, the author, is hard of hearing. Although you can call at (415) 322-6442, he much prefers letters or questions via his BIX conference, Radical.Eye. AmigaTEX costs $200; its printer drivers, $100 each. They're available for Epson FX, MX, and JX 9-pin printers and compatibles, the Imagewriter, for 24-pin Epsons, NEC P6 and P7 printers, and for the H-P DeskJet and LaserJet, plus several PostScript machines. Ask Tomas for the latest list.

Ah, you prefer your lists with numbers? The macro `\begin{enumerate}...\end{enumerate}` will replace the bullets with arabic numbers. If you insert an item or two, LATEX renumbers the entire list—automatically.

It is equally easy to format any quotation. You simply begin and end it with this macro: `\begin{quote}...\end{quote}`, as I now do for one of Oscar Wilde's gems:

> *There is only one thing in the world worse than being talked about, and that is not being talked about.*

LATEX provides a format command for verse (you can parse the lines with absolute freedom) and another for descriptions or definitions, as below:

Bigot: Anyone who vigorously opposes your own views.

Liberal: Anyone who would rather be mugged than shoot the mugger.

Jury: Twelve dull illiterates who swear under oath that they have not read, heard about, or formed any opinions upon a subject that has filled the airwaves and the newspapers for the past four months.

The formatting options above are *environments*. You can use them in any of the major formats—in books, articles, reports, etc. I won't bore you with all the environments; that for mathematics I show later.

This page reveals the nature of LATEX—five major formats within which you can easily invoke specific environments. All the commands you need are "canned" in the form of simple macros. With a bit more work, you can redefine the formats and the environments and save them to disk.

As with all simplifications, LATEX isn't as flexible as TEX when you must create new formats. Although most of TEX is available beneath LATEX, you must experiment to use it. I needed two hours to kludge and waffle the standard LATEX "Article" format to match that of *The Transactor*.

Dimensions

Dimensions in commands may be in millimeters, centimeters, inches, ems, points, or exes (the height of an "x" in the current font), even down to tenths or hundredths. Dimensions of 7.92 cm or .25 pt, for example, are perfectly legal. You may intermix the various dimension standards at will.

WEE VILLAGES OF TEXT
Set in a County of White Space

Default LaTeX Article Format

This page approaches the default LaTeX article format in side margins, spacing and indentation. I cut the enormous top and bottom margins (they otherwise take up one-third of the page) because printing and mailing white space is so expensive that Nick Sullivan, the editor, would have wept. I cheated also by removing the standard "date" line from the title.

This one-column format in 12-point type is called by a prefacing macro: `\documentstyle[12pt]{article}`. Had the beginning macro instead been `\documentstyle[10pt,twocolumn]{article}`, you'd see two columns in the same face used on the previous page. That's simple enough.

A font is one size and style of a specific typeface. This page holds ten *different fonts. Have you priced Bitstream or Adobe fonts lately?*

1 Numbering Sections

By default, LaTeX numbers sections. Subsections appear in smaller type, also automatically numbered. Because the standard section heads devour so much white space, I've removed much of it from the text following.

1.1 Subsections Without Numbers

Section numbering can be turned off with an asterisk in the section macro: `\section*{Section Title}`. On the preceding page, to match the *Transactor*, I redefined the section macro as: `\newcommand{\kap}[1]{{\bf #1}}`, in which `\kap` is the new section command, the number "1" is the first argument (the section title) the macro encounters—and the `\bf` is the standard TeX abbreviation for boldface type in the current font.

I print many macros to give the reader the flavor of LaTeX *and to show how simple and English-like are its commands.*

1.1.1 Who Wants Section Numbers?

Academics, engineers and textbook authors demand them. LaTeX renumbers sections automatically when you remove or insert material. It also carries chapter and section numbers automatically to an optional header on each page.

Smaller and Smaller... Numbered divisions in LaTeX start with Parts and proceed through Chapters, Sections, Subsections, and Subsubsections. Paragraphs and Subparagraphs can be set off with bold-faced captions, as in this and following paragraphs.

General Stuff: I've inserted two marginal notes to demonstrate them. Such notes you place in the text; they appear in the margin *at* the text line holding them. Because it's almost impossible to justify such short lines, I print the notes flush left and right with standard macros.

Enough on the default format. The next page returns to two-column format but holds the standard LaTeX header you may optionally use on the second and subsequent pages of an article. How much time did my LQ-850 need to print this page? About ten seconds more than the same text in 10-pitch letter-quality.

2 Memory & Tabs

Shown below is the memory consumed after I booted with my favorite background programs and a full LaTeX package (my editor, the LaTeX macros, the text formatter, *TeX*, and the *Previewer*), with no text or fonts aboard.[1] Then I brought my draft text into RAM and formatted it there. After several previews of this page, and while the *Previewer* held a formatted page, I checked memory again:

Memory Used:	Chip	Total
Programs alone:	78.7K	812.5K
After drafts:	251.6K	1,366.2K

The table above holds two lessons: First, to run LaTeX and all ancillary programs concurrently, you need a minimum of 1.5 megabytes. If you work on a modest document with many fonts, you require at least two megs. With three megs on line, I've had no problems with long documents.

The second lesson is that tabbing tables in proportional fonts is painful in *any* text-formatting program. Although LaTeX offers several tabbing environments, all seem complex and clumsy. I needed 30 minutes and three previews to tab the table above in the simplest method of all.

3 Fonts & Typography

The terms *fonts* and *faces* are often confused. A *face* names a general design, such as Helvetica or Computer Modern (CM), found in a range of sizes and styles. A *font* is a specific size and style within a face, as in the two 10-point CM fonts, roman and italic, that you read in this paragraph. The fonts of LaTeX, inherited from TeX, differ from those found on laser printers in several significant ways:

1) *Number:* Postscript fonts list at $189 for a set of four (a set might be Times Roman, Times Italic, Times Roman Bold, and Times Italic Bold). Any size can be generated from a basic font. Laserjet fonts are instead bit-mapped; you must buy or generate them at a specific size. For both lasers, the fonts in a document are limited by the size of printer RAM. In contrast, LaTeX provides at minimum 588 different fonts from TeX; of them, 66 hold non-alphabetic symbols. None are downloaded to printer. (Each of the 98 basic printer fonts in TeX you can magnify with `\magstep` commands in at least six steps; $6 \times 98 = 588$.) You'd spend a fortune for that many fonts for a laser printer. To match the printer fonts in the *Previewer* at its many magnifications, you have access to over 1500 screen fonts.

[1] Alas, we face confusion in terms. TeX is the overall text-processing system written by Donald Knuth. It creates a formatted .dvi (device-independent) file with a program named *TeX*, which LaTeX employs also. You must distinguish between TeX and *TeX*. Sorry about that.

2) *Handling:* LaTeX fonts, being in packed format, load from disk into Amiga's RAM swiftly, as needed. (All fonts for this page, *including* symbols, load from floppy in 15 seconds the first time we view or print a page, but remain in RAM until new fonts are required.) A .dvi file created by the formatter, *TeX*, if sent to a 9-pin printer or a 2540 dpi typesetter—of any make—by a TeX printer driver, will call for fonts at the printer resolution before it makes hard copy.

3) *Scaling:* The proportions and strokes of a font must vary as you change its size. As far as I know, only PostScript and *TeX* are designed to do this automatically.

4) *Compression:* The 28 fonts used in this article are cached on drive 1 in sets: one of screen fonts for the *Previewer* in 65K, and a second for my 24-pin printer, in 104.1K.

Fine typography demands more than fonts. Small details make or break a page. Here are a few examples: By default, the *TeX* formatter produces ligatures for 'ff,' 'ffi' and 'fl'—as in efficient, firefly or fisticuffs—and easily generates ægis or œvre. It automatically kerns poor letter spacing (compare kerned AV with unkerned AV, or kerned 'Top' with unkerned 'Top'). An ellipsis... spaces differently than three periods...and can be centered: $a + \cdots + z$.

Good typography demands four dashes: the minus sign $X - Y$, the hyphen (-), the en-dash, as in 'pages 2–4,' and the long or em-dash—as used here.

How many text-processors can compose pages with rubber vertical spaces? In this article, the spacing between lines and paragraphs is elastic. The command `\flushbottom` in the preamble stretches or shrinks all the rubber the same amount until the text is even at the bottom margin.

Although *TeX* automatically hyphenates English, you'll find few hyphens on this page. Unlike other programs, *TeX* parses and justifies entire paragraphs at one time, hyphenating only if it must.

Last, you have exquisitely fine control of all spacing, for *TeX* internally computes in scaled points—65,536 of which equal a printer's point of about 1/72nd of an inch. Bad pages emerge from lack of care, not from lack of control.

4 Symbols: A Cornucopia

I won't bore you with all the symbols fonts, but instead print a sampler. All accents and characters of foreign languages print easily in plain text, from ¿El senor? to garçon and Schloß, including the Greek alphabet from A to Ω in upper- or lower-case. If you don't care for epsilon as ϵ, use ε; choose phi as ϕ or as φ. Printing Greek letters is as easy as π (or Π).

You can $\overline{\textit{overline}}$ or $\underline{\textit{underline}}$ both words and equations, print the general symbols †, ‡, §, ¶, © and £, or use any of the calligraphic letters from $\mathcal{A}$ to $\mathcal{Z}$. You have access to 36 binary operators, among them $\odot$, $\diamond$, $\triangleleft$, $\uplus$, and $\wr$—whatever the heck *that* means...

Symbols of relation total 38—from subset $\subset$ through the familiar $\leq$ and $\geq$ to $\approx$, $\equiv$, $\models$ and $\perp$. The 32 arrow symbols point anywhere: $\leftarrow$, single or double $\Rightarrow$, and in both directions for the confused $\Leftrightarrow$ $\Updownarrow$. You also can throw harpoons $\leftharpoonup$ $\rightharpoonup$, or point diagonally $\searrow$.

The 33 miscellaneous symbols run from Cantor's $\aleph$ to $\flat$ and $\sharp$, through card suits $\clubsuit$, infinity ∞, empty set $\emptyset$, to some you may not know: $\wp$, ∇, and $\forall$. Most you can print in plain text as-is or boxed, as with 'leads to': $\boxed{\rightsquigarrow}$

Symbols are easy to enter: The general ones you escape and name; ¶ emerges from `\P`. Those from math mode you bracket with a $ sign, as in ∞, created with `$\infty$`. You print plain $ by escaping it, so: `\$`. Enough on plain symbols.

5 Mathphobes & Market Tips

Axiom 5.1 *In all English-speaking nations, 96.7% of all children and adults fear mathematics more than hungry crocodiles. Audiences flee screaming to the hills if one simply incants:* $\sqrt[\infty]{\pi}$.

So that the weak and helpless aren't trampled in the rush to the exits, I confine the following summation to one line (and, to shock you, solve the riddle of creation):

Let U_x be any set of equal but opposite universes, Nul be the void (minus time, space or matter), and Q_f be a fudge factor derived from quantum electrodynamics. Then, any number of Universes can emerge from and are equal to Nul—if you don't ask where Q_f came from:

$$\lim_{n\to\infty} \sum_{i=1}^{n} \frac{U_i}{Q_f^n} = Nul \tag{1}$$

The equation above demonstrates one of three math modes. That for math in text I showed in "Symbols." That above is `\begin{equation}`... `\end{equation}`. It centers and numbers each consecutive equation (an option the third mode turns off). You can refer to any numbered equation by label; the one above might be {eq:dumb}. Wherever in text you use that label, LaTeX will substitute the right number for it. That's handy when you revise equations often and their numbers constantly change.

One question remains: Where did **Axiom 5.1**, above, come from? LaTeX offers an environment in which you can label and automatically format anything from hunches, theorems, rules and laws to guesses. Their numbers optionally tie to chapters or sections (as with Axiom, above) or begin with any value, as in the solemn advice below:

Market Tip 1 *When you're asked to buy a stock market letter that will make you rich, be assured that the seller lost a fortune in the market and now intends to live in style on yours.*

Many reviews of TeX and LaTeX conclude that they're costly and great for writing books on math—implying that they're good for little else. I think that's wrong. It frightens mathphobes and fails to account for the cost of built-in fonts.

6 On Uses, Virtues & Flaws

Because LaTeX multitasks robustly (nary a hangup, glitch or Guru), this review is the most favorable I've written since March of '86.[1] Let me now address two questions: 1) What are the flaws in LaTeX? and 2) How does it compare to other text-processing programs?

The Flaws: The LaTeX manual isn't as good as that for TeX. It often talks about complex commands but doesn't provide exact examples. For instance, when I tried to set the width of text, the `\textwidth` macro didn't work until I found a text note that some changes in dimensions must be prefaced with `\setlength`. Many a macro you must dredge out of the text.

Fine Control of typography demands time and patience. For example, LaTeX usually assumes that any of the following symbols ends a sentence: ., ?, :, and !. It then inserts extra space before it begins the next sentence. If you use these symbols *within* a sentence, you avoid the extra space by using a plain one—as I now do for this colon: `...for this colon:\ ` . A backslash followed by a space creates a standard space. Nothing comes free: the price of good typography is time.

Graphics: Although you can make and frame boxes, neither TeX nor LaTeX can import graphics. You can, however, measure artwork to the fractional millimeter, leave space, and paste it in. Tom Rokicki's December '88 newsletter reports that he's revising AmigaTeX to import or include IFF and PostScript files on all printers. In time...

Comparisons: I learned *PageSetter* in about two days, *Pro Page* in ten, got my mind around TeX in about two months, but had the 'canned' formats of LaTeX in hand in two weeks. I prefer LaTeX to *Word Perfect* (WP) for three chief reasons: 1) I can remember macros in English far better than rote combinations of keystrokes; 2) WP remains buggy a year after issue; and, 3) LaTeX is far more powerful and flexible if you work mostly in CLI, as I do. Workbench fans should avoid TeX and LaTeX.

Both programs are ideal for oft-revised documents, such as manuals and handbooks, because text is so easy to change. With the `\input` and `\include` macros you can include old or standard text without formatting it again. Bill Hawes did the manuals for *ARexx* and *WShell* and Charlie Heath the manual for *TxED Plus* with TeX. When I asked Bill Hawes why he preferred TeX, he answered: "Because I like to get down to the basic machinery." (What else can you expect from an old assembly hand?)

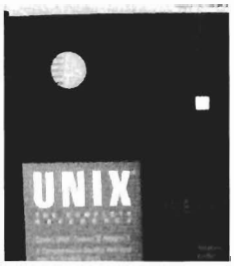
In contrast, John Toebes and the other vintners at the *Software Distillery* (source of BLINK, PopCLI, IconExec, etc.) use LaTeX for all their work—from correspondence to documentation. Any file generated by any bottler can be understood by everyone else—and hammered out on the *Distillery's* H-P LaserJet.

[1] Dick Barnes was an engineer in research and development and backed into computing in the late 1950's as part of his work. After he retired, Dick edited and published *The Amigan Apprentice & Journeyman.* In early 1989, he was forced to abandon it by poor health. He now writes occasional articles about Amiga.

Hard Copy

Introductions to UNIX and C++

by Tim Grantham



UNIX: The Complete Reference System V, Release 3
by Stephen Coffin

Published by Osborne McGraw-Hill
Softcover, 704 pages, $36.95 Can.

Many people believe that the UNIX operating system will soon become *the* standard operating system (OS) for all computers. Sure, there will be always be a need for specialized OSs; the Amiga OS, for example, provides real-time capabilities that standard versions of UNIX cannot match. But nothing has come close to UNIX's broad application; versions exist on almost every kind of computer, from supercomputers to micros. The Amiga is no exception; due to be released this year is the A2500UX, which will run under either the Amiga OS or UNIX, System V, version 3.

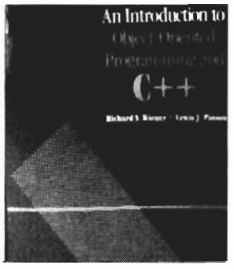
Those not familiar with UNIX will face a steep learning curve. The *UNIX Users' Manual* that comes with every UNIX system does not help the novice much. As pointed out in *UNIX: The Complete Reference*, "It is not uncommon for several users or developers to pore over a single sentence of the manual, discussing (even arguing about) the meaning or implication of a word or phrase... [The manual] is designed to be as concise as possible, to bring the great scope of the UNIX system into a form that experts and near experts can use to find out any of the many specific details of the system. It is very difficult, if not impossible, to learn to use the system from the *User's Manual*, but it is just as difficult to become an expert user without the manual."

UNIX: The Complete Reference, which covers UNIX, System V, version 3, provides the perfect bridge for the neophyte to reach the realm of the UNIX adept. This massive tome provides well-organized, clearly-written information on a vast range of topics. An abbreviated list includes: the shell and shell programming; the file system; system administration; system configuration; running MS-DOS under UNIX; editing and regular expressions; the *uucp* data communications system; the remote file sharing subsystem; and security. The book also provides a tutorial chapter for rank beginners.

Each chapter begins with the fundamentals of the topic, complete with examples, and finishes with a section called 'Going Further', which ventures into the more esoteric capabilities of the system. The reader can gain all the basic user knowledge necessary to administrate a personal UNIX system such as the A2500UX without reading any of the 'Going Further' sections.

The book is intended as an introduction and supplement to the *UNIX User's Manual*. As such, it would probably be more accurately titled *The Complete **User's** Reference*. It provides only the briefest excursions into such areas as UNIX's extensive software development environment, networking and user interfaces.

I make sure this book is by my side whenever I am logged on to a UNIX system. I have experienced no difficulty locating information quickly, via its index, and have so far found it to be completely accurate. Commodore might do well to consider including a copy with every A2500UX sold – it's a perfect complement to the *UNIX User's Manual*.



An Introduction to Object-Oriented Programming and C++
by Richard S. Wiener and Lewis J. Pinson

Published by Addison-Wesley
Softcover, 273 pages, $35.95

C++ (pronounced as either 'C-plus-plus' or 'C-increment') has been available for the Amiga since last year, when Lattice introduced a C++ preprocessor for their v4.01 C compiler. C++ is a superset of the C language, providing object-oriented programming while remaining completely upwardly compatible with C. It's worth noting that release five of AT&T's UNIX System V is being written in C++. See Richard Bielak's article elsewhere in this issue for an introduction to the principal features of C++ and object-oriented programming.

Lattice included the book *An Introduction to Object-Oriented Programming and C++*, by Richard S. Wiener and Lewis J. Pinson, in its C++ package. According to the authors, the book

is intended for students, intermediate-level programmers and software development professionals. They assume the reader has programming experience in C.

Although I do have C programming experience, I had a difficult time with this book. I spent many hours poring over the examples and rereading much of the accompanying text before I began to grasp the essentials. And unlike many readers, I had access to a C++ user willing and able to answer my questions.

Most of the problems with this book arise from a common failing of technical writers: the authors have carried over the artificial and terse nature of the language into their explanatory prose. The book is even organized rather like a C++ program: it begins with an introduction to the object-oriented paradigm (equivalent to a C++ header file), continues with a list of the features that C++ adds to the regular C language (analogous to a C++ *implementation* file), and concludes with several examples of C++ programs (corresponding to the main() functions of C++ programs).

While this approach packs a lot of information into a relatively small number of pages, it doesn't attempt to anticipate any difficulties the reader might have, it is miserly with explanations, and it encourages a very dry prose style. A better approach can be found in Jim Butterfield's articles on assembly language, or David Holzgang's books on PostScript. Both writers have a lively style of writing that provides perspective as well as facts, and that instructs thoroughly without spoonfeeding.

An Introduction to Object-Oriented Programming and C++ also suffers from poor typographical decisions; for example:

> Because the parameter number is passed in as a reference parameter, the simplified rational number is passed back to the object that sent the message simplify. (p. 91)

This sentence makes no sense unless you know that it contains variable names and function names. A better approach would be like this:

> Because the parameter *number* is passed in as a referenced parameter, the simplified rational number is passed back to the object that sent the simplify() message.

The problem is compounded by using variable names that can be mistaken for words being used in a conventional manner; for example:

> A class node is defined with private data next, a pointer to the next node in the singly linked list, and contents, a pointer to the smallest unit of storage, char. Normally one might be tempted to associate the class member contents with a character string. In this case, the private member contents must be interpreted as the starting address of an array of bytes. (p. 101)

This is unnecessarily confusing. Is the next step to define a class node with private data, or is *next* the name of some private data contained in a definition of a class type called *node*? Since the latter case is intended, it should have been printed something like this:

> A class *node* is defined that includes: private data *next*, a pointer to the next node in the singly linked list; and *contents*, a pointer to the smallest unit of storage, char. Normally one might be tempted to associate the class member *contents* with a character string. In this case, however, the private member *contents* must be interpreted as the starting address of an array of bytes.

The final outrage in this regard comes in chapter 7, when the book suddenly switches to the use of italics for variable names:

> This method returns 1 if the character string *wd* is found in the RAM dictionary;... (p. 183)

For a confusing couple of minutes I thought the use of italics was to indicate that the literal string "wd" was being searched for in the RAM dictionary. By the time I realized that *wd* was a variable containing a pointer to a string, they had dropped the use of italics again!

If you think I am being anal retentive about this, consider that the previous example is exactly analogous to confusing the address of an object with its contents. Compilers are not forgiving of such errors; I think it's only fair to expect the authors of books about compilers to use the same precision in their explication.

The exercises at the end of each chapter are much too difficult to expect a student to tackle without guidance or experience; the questions are more suitable as ideas for teachers of C++. For example, after the brief introduction to the basic concepts of object-oriented programming, the first exercise is: "Partition a software development problem of your choice into classes, sub-classes, objects, and methods at the highest level of design." The reader *might* be able to do this after having read the entire book and carefully examined the example programs; the authors certainly do not spell out the reasons for the way they decomposed the problems addressed by the sample programs.

There are rewards to be gained from *An Introduction to Object-Oriented Programming and C++*, for the reader with patience. The example C++ programs are very well constructed, and implement such useful capabilities as: the ability to use regular arithmetic and logical operators with a rational number type; a 'super-fast' spelling checker; a generic list

class that can store and manipulate different types of data transparently; and an interactive function evaluator. A bank teller discrete event simulation demonstrates the language's power as a modelling tool.

But while the book does provide an comprehensive overview of the language, and illuminating examples of C++ source code, poor choices in the book's typesetting and organization produce needless frustration for the reader. When combined with the terse style of writing, the result makes learning C++ from this book a little like trying to learn electronics by puzzling over a poor photocopy of a television set schematic.

When editors Tim Grantham and Chris Zamara first discussed the subject of this review, it was hard to believe the two were talking about the same book; Zamara did nothing but rave about its excellent organization and logical presentation of the new language concepts. Since topic organization is highly subjective, we include Zamara's point of view.

I think many programmers share my seemingly random method of learning: a new piece of information leads off on a tangent, whose exploration reveals a new layer of information and accompanying tangents. The Wiener/Pinson C++ book surprised me by being one of the few books that presented the information I wanted to know, in the sequence I wanted to learn it. Starting at the beginning of the book, I was pulled through to the end without great effort, at which point I knew enough about C++ to begin coding. Unlike many texts on programming languages, the book is not dull, and in fact makes learning the foreign programming concepts and language dialect something of an adventure.

Yes, the writing style is terse, and the typestting gaffes can lead to a few double-takes, but I found the organization of the book ideal, providing you are already quite comfortable with C. It starts by introducing object oriented programming in general, which is necessary to grasp before dealing with a specific implementation of those concepts: it answers 'Why?' before you begin learning How. The next section shows how C++ enhances C, first in 'small ways' (new C-like features), then in 'large ways' (object oriented features). This section is like introducing the pieces of the puzzle before showing you how they fit together. It also serves as an appetizer: it had the effect of drawing me further into the book in order to learn how to apply these strange and wonderful new tools.

Language concepts are introduced in the following sections, increasing in difficulty and building on previous material. Explanations are sparse at times and even confusing, as Tim's examples point out, but the abundance of relevant code examples helps greatly in clarification. By the time you get to the sample programs in the last chapter, the once-bizarre C++ code actually makes sense!

As an easy step-by-step tutorial, the book may disappoint. It's more of a quick-moving exploration, and if that's your learning style, it can be an enjoyable - and rapid - way to learn C++.

flickerFixer, by MicroWay, Inc.
Suggested retail price: $595 US

Review by Tim Grantham

Those who have a low tolerance for the Amiga's interlace 'twitter' now have recourse to *flickerFixer* (FF), a board that plugs into the video slot of an A2000 and provides a de-interlaced output to any VGA-compatible monitor.

Because FF generates twice the number of vertical sync pulses per second as the Amiga's standard video output, it cannot be used with a standard Amiga monitor. It requires a monitor that can accept a vertical scan rate of about 31.5 KHz. All VGA-compatible monitors (VGA stands for 'video graphics adaptor', which, in the PC world, provides the emerging industry standard for graphics), such as the NEC Multisync, the Mitsubishi DiamondScan, the Zenith FTM and a host of others, work at this frequency.

We tested FF with a Nanao Flexiscan multisync monitor. The term 'multisync' usually means that the monitor will operate at a variety of scan frequencies, including that of the Amiga.

Installing the board was a simple task: open the 2000's case, plug it into the video slot and close the case. No software installation is required, although Microway does provides a disk containing utilities for positioning the display on the screen (this does not alter the display on a standard monitor, which can be connected at the same time), and for checking video alignment. We had to use an adaptor to connect the monitor to the FF – not a difficult task because the FF provides a standard, female 9-pin DIN output connector.

The FF/Flexiscan combination provided a gorgeous, rock-solid display. *Professional Page* looked... professional. *UEdit* and other text editors with interlace support displayed more than 40 rows of lovely, clean text. Running a game with high-speed animation produced no image breakup that I could see. It should be noted that the monitor itself accounted for much of the quality of the display, with its fine dot pitch and saturated colours (the blacks are *really* black).

There are two drawbacks to FF: it takes up the video slot, making it unavailable for other video hardware, such as a frame-grabber or genlock; and, more seriously, it costs a lot of money – $595 US for the FF (the same as a new A500), plus the cost of a multisync monitor, which starts at about $500 US and can go up past $2000 US. Still, if you are spending a lot of time working with interlace in a professional environment, this product may well pay for itself by reducing operator stress and increasing productivity.

FREE: Amiga Demos

Have you ever wished you could view a program before you purchased it? Do artist conceptions and written descriptions really give you a good idea about the program you are buying? No!

We will send you actual manufacturers demonstration programs for you to view, on your own computer, in your own time. The gimmick is: if you like the program, buy it from us! We're even competitively priced!

Send us your name and address, we'll send you an ***Amiga Demo FREE!***

We ship anywhere in the World!

Forethought
Innovatiove Software Distribution
P.O. Box 1221
Lewiston, NY 14092

68000 DISASSEMBLY

OTG Software

ANNOUNCING . . .
DSM VERSION 1.0d

DSM is a full-featured disassembler for the Amiga. Check out these features and you'll see why programmers agree, "DSM is the best disassembler currently available for the Amiga, bar none."

- **DSM disassembles any Amiga program that does not use overlays.**
- **DSM produces assembler-ready output that is 100% compatible with the Amiga assembler, assem. Excellent results can also be obtained using the C.A.P.E. assembler from Inovatronics.**
- **DSM features true text detection.**
- **Improved disassembly is supported using the built-in expert system mode.**

If your local Amiga dealer doesn't carry DSM, send check or money order to:

OTG Software
200 West 7th Street Suite 618
Fort Worth, TX 76102

Price
$67.50 US

TX residents add 7.25% sales tax.

COLOR RIBBONS & PAPER

RIBBONS: Red, Blue, Grn., Brwn., Purple, Yel.

Ribbons	Price Each	Black	Color	Heat Transfer
Brother M1109		4.95	5.95	7.00
C. Itoh Prowriter Jr.		7.00	9.00	–
Citizen 120D/180D		5.00	6.00	7.95
Commodore MPS 801		4.50	5.25	5.75
- MPS 802/1526		6.25	7.25	–
- MPS 803		4.95	5.95	7.00
- MPS 1000		3.95	4.95	6.75
- MPS 1200/1250		5.00	6.00	7.95
- 1525		6.50	8.00	–
Epson MX80/LX800		3.75	4.25	6.75
Okidata 82/92		1.75	2.25	4.50
Okidata 182/192		6.50	7.50	–
Panasonic K-XP 1080		6.75	7.75	–
Seikosha SP 800/1000		5.25	6.50	7.95
Star SG10		1.75	2.25	4.50
Star NX10/NL10		5.00	6.00	7.95
Star NX1000		4.50	5.50	8.00
Star NX1000C – 4C		–	10.75	–

COLOR PAPER

BRIGHT PACK—200 Sheets/50 each color: Red, Blue, Green, Yellow. 9 1/2 x 11 — $11.90/pk.

PASTEL PACK—200 Sheets/50 each color: Pink, Yellow, Blue, Ivory. 9 1/2 x 11 — $11.90/pk.

T-SHIRT RIBBONS (Heat Transfer) — Red, Blue, Grn., Brwn., Purple, Yel., Blk. Call For Price & Availability.

COLOR BANNER BAND PAPER — 45 ft./roll—$9.95/ea.

For ribbons & paper not listed above, call for price & avail. Price & spec. subject to change w/o notice. Min. order $25.00. Min. S&H $3.50 min. Visa, MC, COD.

RAMCO COMPUTER SUPPLIES
P.O. Box 475, Manteno, IL 60950 U.S.A.
1-800-522-6922 • 815-468-8081

A CSA TURBO 68020 FOR ONLY $399?

Hard to believe isn't it? CSA broke the price barrier in 32 bit technology. Now you don't have to settle for a far less capable 68000 accelerator, you *can* have affordability, capability, and speed in one easily installed package.

"1000% faster than my stock Amiga™?"

CSA's new 68020 *Midger Racer*™ flies through MaxiPlan™ spreadsheets over 60% faster than the standard Amiga™, and over 50% faster than 68000 based accelerators. With the addition of a 12 MHz 68881 co-processor, the *Midget Racer*™ can crunch 2 Million FLOPS over 1000% faster than a stock Amiga™.

"600% faster than a 68000 accelerator?"

In fact, programs like Sculpt & Animate 3D or 4D™, Draw 2000™, and X-CAD™ have been written to directly access the 68020 & 68881, and may not even *run* with a 68000 based accelerator. And with a 12 MHz 68881, CSA's *Midget Racer*™ is up to 600% faster than a 68000/68881!

"All that for only $399?"

No, there's more! It's also easily installed into ***any*** Amiga, supports an '881/2 up to 33 MHz, and it's available now. Don't settle for second best, call your Amiga dealer or CSA today!

CSA. We set the standards in performance.

CSA

Computer System Associates, Inc.
7564 Trade Street
San Diego, CA 92121
619-566-3911 FAX: 619-566-0581

Trademarks: Amiga: Commodore-Amiga, Inc.; MaxiPlan: Oxxi, Inc.; 68000/020/881/882: Motorola, Inc.; Sculpt & Animate 3D/4D: Byte by Byte, Inc.; Draw 2000: Aegis, Inc.; X-CAD: Taurus, Inc.; Midget Racer: Computer System Associates, Inc.

Lattice C Compiler Version 5.02

Once the underdog, now Lattice has the lead

by Nick Sullivan

Lattice C Compiler v5.02
Lattice, Inc., 2500 S. Highland Avenue,
Lombard, Illinois 60148 USA.
Price: $300 US. For owners of Lattice 4.0 and later, $75 US. For owners of Lattice compilers prior to 4.0, $100 US. For owner of any version of the Manx Aztec compiler, $150 US.

Not all that long ago, Lattice C was an embattled underdog in the Amiga C compiler wars. Manx Software Systems' Aztec compiler was faster, smaller, and produced better code; Manx also incorporated a debugger, an editor, an assembler, a *make* utility and other supporting software with their compiler in reasonably-priced bundled packages.

With all that going for it, the Aztec compiler seemed to have a solid lead in user acceptance. And Lattice's initially flaccid commitment to the Amiga made it appear unlikely that they would be able to make up the lost ground.

They have done so all the same. After their indifferent start, Lattice has steadily gathered momentum. They have looked at Manx's strongest selling points, and one by one have incorporated them in their own product, often in improved form. At the same time, they have provided major new features that Manx does not yet provide: ANSI conformance and more advanced code optimization, for example. It is now Manx's turn to play catch-up.

(Lattice's current version number is 5.02, and the compiler is referred to as such in this review. Apart from bug fixes, though, it is the same product as versions 5.0, the last public release. If you are a registered owner of 5.0, contact Lattice for a free upgrade.)

What's in the Lattice package?

We have come a long way from the slim white manual and single floppy disk of the earliest Lattice compiler for the Amiga. Today's product lives on five floppies and two thick manuals - physically, it is one of the largest Amiga software packages around.

At the core of the package is the compiler itself, and the linker, *Blink*. The compiler operates in two phases. The first, which uses either the *lc1* (small) or *lc1b* (large) programs, preprocesses the source code and compiles it into an intermediate form known as a 'quad file'. The larger of the two programs (about 120K instead of about 100K) includes support for function prototyping and for generating a cross-referenced listing of the source code. The second compilation phase, embodied in the program *lc2*, is responsible for translating the quad file into a linkable object file. A fourth program, *lc*, simply invokes the two compilation phases in succession - it works as a kind of shell to simplify user interaction with the compiler. For simple projects, it can also be used to invoke the linker automatically after compilation.

A fifth program, the global optimizer *go*, may optionally be included in the compilation sequence. It takes the quad file generated by the first compiler phase as input, and creates a modified quad file as its output.

The other major components of the Lattice package are the text editor, *lse*; a make utility, *lmk*; an assembler, *asm*; a source level debugger, *cpr*; and an execution profiler, *lprof* and *lstat*. A good assortment of miscellaneous utilities is also provided; these include an object module disassembler, a library maintenance utility, versions of *grep* and *diff*, and a program that tokenizes header files for faster compilation.

Registerized parameter passing

Parameters to C functions are normally passed on the stack, since it provides reasonably copious and efficient storage, and is readily addressable by relative modes suitable for recursion. In assembly language, function parameters are more often passed in CPU registers, which are even more efficient.

One of Lattice 5.02's most significant innovations is to extend this efficiency advantage to C programmers. This is done through a compiler option in a clever and fairly transparent way. When the option is in force, the first two pointer parameters to a function are passed in address registers a0 and a1; the first two integer parameters are passed in data registers d0 and d1. All other parameters are passed on the stack as usual.

In order to make this method work, Lattice has adopted a special internal convention for the names of functions compiled

with registerized parameters. In most circumstances, this has no effect upon the programmer; if it does cause problems in some specific instance, the feature can be turned off on a function by function basis as required.

That small caveat aside, passing parameters in registers makes a lot of sense: you get significantly smaller, faster code at no programming cost - an unusual combination!

ANSI conformance

Lattice 5.0 comes very close to complete compliance with the new ANSI standard for the C language. According to the documentation, there remain only "a couple of areas where we have not reached full compliance:

"• Not all of the tri-graph sequences intended for support of incomplete keyboards are implemented.
"• Some obscure keyword orderings in declarations are not correctly parsed."

Assuming the word 'obscure' was aptly chosen, these are not major omissions. The ANSI features that *are* implemented include (among others) a much-extended preprocessor, support for new type keywords like *const* and *volatile*, and full new-style function prototypes and declarations, such as:

```
void main (int argc, char *argv[])
```

rather than:

```
void main (argc, argv)
int argc;
char *argv[];
```

For more information on the ANSI innovations to C, refer to Eric Giguère's article, "The ANSI Standard For C" (*Transactor for the Amiga*, Volume 1, Issue 3).

Other compiler features

The compiler has been enhanced in a number of other ways - more than there is space to list here. Many small code optimizations have been added, such as delayed stack clean-up (resetting the stack pointer after a function call only when necessary) and condition code tracking (to eliminate superfluous tests).

Lattice has mimicked Aztec C's useful ability to handle precompiled symbol files, and goes one better by allowing more than one symbol file to be included in the same compile. It is also now possible to compile specifically for each of the 680x0 processors, and for the 68881 and 68882 math co-processors.

The global optimizer

The *go* program, which is normally run indirectly through a command line option to *lc*, is capable of major transformations on the intermediate file generated by the first compiler pass.

One effect of using *go* is that your use of the *register* keyword in variable declarations is completely ignored in favour of the optimizer's own judgements. Moreover, variables may be held in registers for only part of their lifetimes, then be moved onto the stack to free register space for other values. This free-form approach clearly has a lot of potential for creating more efficient code, and should lead to substantial gains. In the current version, though, an inspection of the generated code will show a fair amount of unnecessary traffic into, out of and between registers. That will presumably be reduced as the optimizer is further improved in future releases.

The Lattice manual lists numerous other optimizations that *go* applies to expressions and loops. For example, sub-expressions that are re-used within an expression (e.g. *i+1* in *a[i+1] = b[i+1]*) will be calculated only once, and the result cached, in the executable program; invariant assignments within a loop (e.g. *for (x=12, i=0; i<10; i++) k=x+10;*) will be moved outside the loop; and multiplications involving a loop control variable will be reduced to additions where possible.

In another class of optimizations, dead assignments (assignments to variables that are not subsequently referenced) and unused automatic variables are eliminated from the functions in which they occur.

It is not really meaningful to benchmark the optimizer's performance, which depends heavily on the style and the quality of the input code. It will almost always help to some degree, however, and could make a dramatic difference in some instances. Nevertheless, the optimizer has one drawback for everyday use - it is (understandably) quite slow. For most development work, it is probably best to compile normally until the program is close to completion, and only bring in the optimizer at the final testing stage.

Bugs and gotchas

The original Lattice 5.0 was not particularly reliable, and was quickly superseded by the current 5.02 release. Though 5.02 is much more stable, Lattice still has some work to do before they can claim a rock-solid product.

My first serious work with 5.02 was to Latticize a fairly large program written for Manx C. That job took two or three days: C is C, but Manx isn't Lattice, and K&R isn't ANSI. By the time I had persuaded every converted module to compile, Lattice's function prototype checking had caught a couple of errors involving wrong or missing parameters. The compiler also turned up one rather horrifying instance of using the value of an uninitialized auto variable in an expression. So far, so good.

I now had a gaggle of object modules, but not yet an executable program. Getting one was harder than I anticipated. *Blink* is one of the best parts of the Lattice package - fast, fairly small, easy-to-use - but its use can require some thought, particularly because there are numerous *lc* options that affect

linking. The compiler gives you a lot of control over data storage: you can specify that data will be referenced either at absolute addresses, or relative to a data segment pointer in CPU register A4; you can separately specify that the data must go in Chip or Fast memory. Moreover, any of these specifications may be applied to a whole module but overridden for a particular data item. The upshot is that you have far more flexibility in this area with Lattice than with Manx - so much flexibility, in fact, that you can end up specifying nonsense if you don't think about what you're doing. I spent about a day and a half specifying nonsense.

In the end, I got my executable program, and was impressed - it came out about 15 per cent smaller under Lattice than Manx. That difference alone justified the conversion effort.

The only problem was that some formerly reliable parts of the program now failed to work properly, or gurued. It took a couple of long sessions of studying disassemblies to isolate and work around these bugs, which in each instance were introduced by the compiler itself. It probably won't be long before more reliable releases of the compiler appear, since Lattice has a good record for updates and fixes. For the time being, though, caution is advised: the 5.02 and earlier versions can introduce hard-to-trace errors into correct programs. The many strong points of this compiler make it attractive for serious development work, but even innocent-looking code should be tested very thoroughly to ensure that it hasn't been broken by some obscure compiler quirk.

(Just as we go to press, we have word that a 5.04 patch will be available to registered users on May 1 - about the same time this magazine appears on newsstands. That release should correct all bugs reported to date.)

Other programs

The main attraction of Lattice 5.02 lies in the core programs: the compiler, the linker, and the *cpr* debugger (which is broadly similar in capability to Manx's *sdb*, though very different in detail - see Vic Wagner's "BreakPoint" article in *TransAmi*, Volume 1, Issue 6 for the details). Like Manx, though, Lattice nowadays rounds out their package with a very full set of supplementary utilities; I'll briefly look at a few of them here.

Asm is a full macro assembler, capable of generating code for any of member of the 680x0 processor family. *Asm* is not as fast as Manx's *as*, nor as quirky. As noted above, it has the special advantage within the Lattice environment of supporting the at-sign prefix for functions that use parameters passed in registers.

Lse is Lattice's screen editor, apparently much enhanced from the old versions (that - rumour says - Lattice's own people refused to use). If you already have a text editor you're happy with, *Lse* probably won't seduce you away from it; if you don't, *lse* is probably at least adequate, and maybe more - editors are notoriously a matter of taste. One bonus that this editor *will* give you is an 'integrated environment' - the ability to compile, link and correct errors from within the editor without stepping outside into the CLI.

The execution profiler (*lprof* and *lstat*) is a handy utility that can be a big help in locating and eliminating processing bottlenecks in your programs. Your executable program (which should be linked with symbols included) is run under the profiler with:

```
lprof myprog
```

Lprof samples the position of the PC (the CPU's program counter) at frequent intervals during the run, and records the samples in a file that can later be interpreted and made human-readable by *lstat*. The profile information you get back will give you a function by function account (or even line by line, if you need it) of where your program is spending its time. This shows you immediately where to concentrate your optimizing efforts if you want your program to run more efficiently.

The manual

The first totally satisfactory compiler manual is still waiting to be written, but Lattice's effort here is creditable. One has to applaud their attempt to keep the huge mass of information presented under control: the two-volume manual comes in six big sections, each with its own index, and there is a master index at the end incorporating each of the others. The general organization of the text is good, and charts and diagrams are sometimes used to centralize or clarify the information.

Finding a particular piece of information can still be a frustrating exercise, all the same. I think that what's missing is cross-reference information between the various sections of the manual. For example, various aspects of using the compiler are discussed in several different places; it would often be easier to find the crucial fact one needs if there were more "see page *n* for further information" signposts included.

And the winner is...

The clear winner in the C compiler wars is, as usual, the Amiga programmer, who has seen both the Aztec and Lattice products go from strength to strength since the machine was introduced.

Lattice 5.02 is undoubtedly the strongest C compiler the Amiga has yet seen. If you're now trying to decide which of the two available compilers is right for you, Lattice is probably the way to go. If you own Manx already, should you switch? Not necessarily. Aztec 5.0 (Manx is reportedly planning to skip a 4.0 generation, for psychological/marketing reasons) is scheduled for release later this year, and is expected to offer ANSI compliance; whether it will be able to vie with other Lattice features is not yet known, but it's reasonable to assume

that Manx will try to remain competitive. For now, though, the ball is clearly in their court - Lattice has taken the lead.

A very brief example

We've always felt that the ideal compiler review should include a reasonable number of examples of generated machine code, but that's a luxury space and time don't allow. Just to give you *some* idea of what the Lattice 5.02 compiler might do with your source code, though, here is the result of compiling a very small (and quite pointless) C program in three ways

The first is a default (no options) compilation, except that the automatic generation of stack checking code has been turned off. The second uses 'registerized function parameters', and the third adds the manipulations performed by the global optimizer. Code size in bytes and execution time in cycles is shown for each version. Here's the source code for this tiniest of tests:

Listing: test.c

```
/*
   test.c

   lc -cf -v -onormtest.o        test     (normal compilation)
   lc -cf -v -oregtest.o  -rr    test     (registerized function parameters)
   lc -cf -v -oopttest.o  -rr -O test     (use global optimizer)
*/

void main(int, char **);      /* function prototypes */
int  puts(char *);
void exit(int);

void main (argc, argv)
int argc; char **argv;
{
int n;

   n = 200;                   /* superfluous assignment */

   if (argc != 2)
      exit(n);

   puts(argv[1]);
}
```

Default compilation - 46 bytes, 228 cycles

```
                                          bytes   cycles

MOVEM.L   D6-D7/A3,-(A7)          ;  4      32
MOVE.L    0010(A7),D7             ;  4      16
MOVEA.L   0014(A7),A3             ;  4      16
MOVEQ     #64,D6                  ;  2       4
ADD.L     D6,D6                   ;  2       8
MOVEQ     #02,D0                  ;  2       4
CMP.L     D0,D7                   ;  2       6
BEQ.B     001E                    ;  2      10
MOVE.L    D6,-(A7)                ;  2      12
JSR       _exit(PC)               ;  4      18
ADDQ.W    #4,A7                   ;  2       4
MOVE.L    0004(A3),-(A7)          ;  4      24
JSR       _puts(PC)               ;  4      18
ADDQ.W    #4,A7                   ;  2       4
MOVEM.L   (A7)+,D6-D7/A3          ;  4      36
RTS                               ;  2      16
```

With registerized function parameters - 38 bytes, 180 cycles

```
                                          bytes   cycles

MOVEM.L   D6-D7/A3,-(A7)          ;  4      32
MOVE.L    D0,D7                   ;  2       4
MOVEA.L   A0,A3                   ;  2       4
MOVEQ     #64,D6                  ;  2       4
ADD.L     D6,D6                   ;  2       8
MOVEQ     #02,D0                  ;  2       4
CMP.L     D0,D7                   ;  2       6
BEQ.B     0018                    ;  2      10
MOVE.L    D6,D0                   ;  2       4
JSR       @exit(PC)               ;  4      18
MOVEA.L   0004(A3),A0             ;  4      16
JSR       @puts(PC)               ;  4      18
MOVEM.L   (A7)+,D6-D7/A3          ;  4      36
RTS                               ;  2      16
```

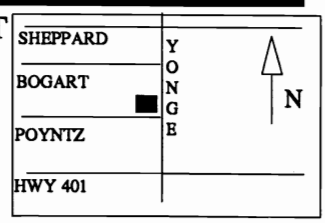
With global optimizer - 40 bytes, 174 cycles

```
LINK      A5,#FFF8                ;  4      16
MOVE.L    D0,FFF8(A5)             ;  4      16
MOVE.L    A0,FFFC(A5)             ;  4      16
SUBQ.L    #2,D0                   ;  2       8
BEQ.B     0018                    ;  2      10
MOVEQ     #64,D0                  ;  2       4
ADD.L     D0,D0                   ;  2       8
JSR       @exit(PC)               ;  4      18
MOVEA.L   FFFC(A5),A0             ;  4      16
MOVEA.L   0004(A0),A0             ;  4      16
JSR       @puts(PC)               ;  4      18
UNLK      A5                      ;  2      12
RTS                               ;  2      16
```


CANADA'S BEST SELECTION OF
AMIGA - C64 - C128 SOFTWARE
COMSPEC HARD DRIVES AND BERNOULLI DRIVES
AVAILABLE FOR
THE AMIGA 500 AND 2000

Electronics 2001
COMPUTER PRODUCT CENTRE

4736 YONGE STREET
WILLOWDALE ,ONT
M2N 5M6
TEL: (416) 229-2700



AUTHORIZED COMMODORE DEALER SINCE 1979

New Store!
Fisher's
Computers & Software
Authorized Amiga Dealer
8005 Archibald
Rancho Cucamonga, CA 91730
(714) 987-1662
Serving The Inland Impire

Cygnus Ed Professional

A fast and powerful text editor

Review by Steve Ahlstrom
Copyright © 1989 Steven Ahlstrom

CEDPro, from ASDG Inc.
Multi-document text editor; requires 512K RAM, AmigaDos 1.2 or later. Suggested retail price: $99.95 US.

A text editor is not a word processor. In general, word processors are mostly concerned with the layout of text, the final look on paper. Word processors are for writers. A text editor is for programmers and editors: it is a tool for entering text, lots of it, easily and quickly. So, while word processors and text editors allow the entering and manipulation of text, in most cases that is where the similarity ends.

A text editor does not have or require all the features of a word processor. There is no footnoting, table of contents generation, sophisticated graphic and graphic text handling, WYSIWYG display, print preview, or a myriad of other features a word processor may have. The converse is also true: a text editor has features that word processors do not offer. ASDG's Cygnus Ed Professional (CEDPro) is a very powerful and feature-rich text editor.

The nicest feature of CEDPro is that it is *fast*. Screen scrolls are faster than the eye can follow (up to 30 thousand characters per second - the scroll rate is selectable); and the search function scans 100 thousand characters per second.

System selectables

CEDPro is one of the most configurable text editors available for the Amiga. It can be run from the Workbench or from a custom screen. Screen width and height can be adjusted beyond the Amiga's normal 640x200/640x400 resolution range. This is of great benefit for those running PAL or 'morerowed' systems as well as the newer high-resolution monochrome monitors.

CEDPro does *not* allow for user selected fonts, nor does it work with Dave Haynie's SetFont program. User-selectable fonts is not a necessity in a text editor but it would be a nice option to have - especially to have the ability to use Topaz 11 when in interlace mode.

Practically all programs that feature a text 'scroll bar' place the gadget on the right side of the display. CEDPro gives you a choice. The scroll gadget may be placed on the left, right, or not present at all.

There is a 'hot start' option; one way to make use of this is to invoke CEDPro from your startup-sequence. If you do this, CEDPro won't bring up an editor window, but will be in memory waiting for you to activate it. Productivity is increased by not having to wait for the program to be read in from disk each time you need it. When in its 'dormant' state CEDPro reduces its memory requirements to a minimum and does not consume CPU time.

When it is in its dormant state, CEDPro can be awakened in one of two ways: with a hot-key sequence, or the supplied Ed program. Ed takes parameters (in this case, names of the files to be edited) and passing them to CEDPro. CEDPro 'wakes up' and loads in the specified files.

The Amiga, being a multitasking system, often has many different tasks running at any given time. These tasks are assigned 'priority' levels. Most user tasks run at a priority of 0. The higher the priority, the more CPU time a task will get. Tasks running at a lower priority will not interrupt the higher priority task. They get *their* chance to run when the higher priority task is waiting for an event to occur. CEDPro provides the ability to change its own priority level. The priority at which the program will run can be inherited from the priority of the CLI that called it.

As an option, CEDPro will create icons for your data files. For Workbench users this is a nice touch. By using the 'Set Icon Tool Name' feature, the default tool can be specified. The icons will automatically be saved 'knowing' the correct path to use for the location of CEDPro on your disk. For CLI users, just

turn this option off so as not to waste disk space for unwanted **.info** files.

CEDPro uses the default colour set of your Workbench preferences. Should you specify the program to run on a custom screen, the arrangement of these colours may be changed. There are 24 possible colour combinations for background, text, status lines, and so on. The colours may be cycled through the variations until a combination that best fits your needs is found. When running the program in a Workbench window the colour-change option is not activated.

Have you ever been working on a document only to get hit with a power surge or power failure? How long ago was the last save of your document? Unless you have saved to disk in the last few minutes, you could find yourself in the situation of having to start all over again. CEDPro provides a safety valve for these occurrences. Using the 'Autosave' feature, you will be prompted every *n* minutes to save work in progress. The frequency between the prompts is selectable with the 'Set Timer' command, and can be from 5 to 99 minutes.

Another approach to safeguarding your data is the 'Do Safe Saves' option. When active, this provides an increased level of security at the expense of a little time. When you save an already-existing file, the original version is first renamed. The edited version is written to disk, then the temporary file is deleted. Should a problem occur during the write to disk the original (unedited) file is still available. Using this option will slightly increase the time necessary for file saves, and requires that the destination disk have enough free room to, temporarily, store two copies of the file. When your data would be very difficult to recreate, the extra time is a very fair price to pay.

The environment settings you choose can be stored in a file that will automatically be loaded by CEDPro when the editor is run. This 'ceddefaults' file contains all the settings described above. Set up the program options once, save the defaults file, and the program will be configured to your liking the next time it is run. CEDPro allows for multiple default files. If you are a C programmer, for instance, your source modules normally will have a **.c** extension. You may configure the editor for your ideal programming environment. When saving your preferred environment for editing C source, use the **.c** extension for the ceddefaults file (that is, 'ceddefaults.c'). The next time a **.c** file is edited, 'ceddefaults.c' will be the environment file loaded. Similarly, should you have the editor configured differently when editing text files, attach the **.txt** extension to 'ceddefaults' and have that environment become active the next time a **.txt** file is edited.

Getting to work

One of the most powerful features of the Amiga, with the addition of ARexx by William Hawes, is the ability of one application to communicate with another application. CEDPro fully supports ARexx. Using this facility, you are able to run another application from within CEDPro, manipulate data, then import that data into CEDPro. With careful planning, it is possible to use ARexx scripts (programs) to configure CEDPro as an integral editor for the programming language of your choice, then compile/assemble from within CEDPro. Should an error occur, the offending file can be automatically loaded into the editor with the cursor positioned on the error line. This capability gives CEDPro a tremendous amount of utility. Calls to ARexx scripts are assigned to function keys, and the assignments can be saved in the 'ceddefaults' configuration file.

CEDPro allows up to ten files to be loaded and visibly displayed at once. These may be unique files or views into the same file. They are displayed in an area within the editor window; this is known in CEDPro parlance as a 'view'. A view is much like an Intuition window, only program-maintained, and not sizable horizontally. Each view has its own title bar, display area, and scroll gadget. All files loaded into the editor are visible. You can cut from one view and paste into another. Each view is independent of any other. You may vertically expand a view to see more lines of text. Using the 'Auto-Expand Views' feature, you may have the views automatically expand to the fullest possible display area when they are clicked upon.

A frequent text editor user demands that the editor have a user-definable macro facility. CEDPro's macro functions are very complete. You may assign a macro sequence to a single keystroke or to a string of keystrokes (similar in fashion to uEmacs). Macros may be assigned to any numeric keypad key, function key, or main keyboard key. Using combinations of the ALT, CTRL, and SHIFT keys, different macros can be assigned to the same key. Macro sequences may be saved and become active the next time you run the editor.

CEDPro contains all the 'normal' features and functions one expects to find in a powerful text editor. Most programmers want an auto-indent return key: when ENTER is pressed, the cursor will advance one line and be placed in the same starting column position as the line above. This makes program formatting quicker to accomplish. Tab tables are supported as well as user-definable tab sizing. Tabs can be 'real' or simulated with spaces. Cut/Copy/Paste and forward and reverse search and replace are all supported, as well as the ability to jump to a specific line. Specific locations within the file can be marked as a reference point; having set one of these, you can then jump the cursor back to it instantly from anywhere else in the file.

Although CEDPro is not, as I pointed out above, a word processor, it does offer a number of 'word processing' features. These include word wrap, multiple column support, keymap support, layout mode, the ability to show embedded control codes, and more. Although the word processing features are minimal, they can be handy, especially for small jobs.

In summary, CEDPro is a powerful, high-end text editor with little competition. It is easily able to meet the demands of the frequent text editor user. And thanks to the ARexx interface, it can grow along with your own needs.

Manipulating AmigaDOS File Names

Simple functions for everyday jobs

by Nick Sullivan

Just about every program you write is going to have to manipulate file names in one way or another. Very often you'll find yourself creating the same simple functions over and over again to do things like:

- Split a path name into directory and file name components
- Recombine the directory and file names into a full path
- Replace the current file name extension with a new one

You must be careful, when using the functions here, to supply buffers of sufficient size to handle the longest names you might encounter. File names are restricted by AmigaDOS to 30 characters, and path names are restricted to 255 characters.

There are some hazards relating to name length, even with buffers that meet the size requirements. The most obvious is that your name buffers must be sized not just to DOS's requirements, but to those of whatever input mechanism you are using; that is, don't pass a 31-byte file name buffer to a requester that uses a 40-byte string gadget!

Secondly, remember that it is possible to concatenate a legal file name with a legal directory name and create an illegal name for the total path. This will happen only if the combined length of the names is greater than 255 bytes - not an everyday occurrence, but a conceivable one.

More subtly, you can actually create files or directories whose full path specifications exceed the AmigaDOS limit. This is made possible by the use of logical assignments. To see this in action, put an empty disk in df0:, and make the following little Execute script:

```
makedir y:abcdefghijklmnop
assign  y: y:abcdefghijklmnop
```

Begin by assigning **y:** to df0:, then Execute the script at least 16 times. It won't fail, even though we have exceeded the 255-character limit for a total path name. In fact, there is probably nothing to stop you from carrying on into deeper and deeper levels of directory nesting to the limit of your patience. If you now offer the name 'y:' to a function that expands path names fully (many file requesters do this, for instance), you are likely to crash either your own program or AmigaDOS itself.

Whether you want to test for extreme conditions like this is up to you - in actual practice, the danger is very remote - but it *is* a good idea to at least be aware that problems can occur.

ParsePath

The first of the tasks listed above is that of splitting a full path name into its directory file name components. The method for doing this is simple: check backwards from the end of the path name until a '/', a ':', or the beginning of the name is encountered, and split the name at that point. This implementation depends on a function called *rindex()*, which scans a string backwards for a particular character, and returns a pointer to that character, or NULL if the character is not found. The Manx function library (*c.lib*) contains *rindex()*; Lattice, however, does not. Here is an assembler version for use with Lattice:

```
* From C    : char *p = rindex(char *s, int c)
* Assembler:   d0.l              a0.l    d0.b
          XDEF    _rindex
          XDEF    _xrindex
_rindex:  move.l  4(a7),a0    ;C entry point
          move.w  8(a7),d0
_xrindex: suba.l  a1,a1       ;assembler entry point
rnx_wl    move.b  (a0)+,d1
          beq.s   rnx_ex
          cmp.b   d0,d1
          bne.s   rnx_wl
          lea     -1(a0),a1
          bra.s   rnx_wl
rnx_ex    move.l  a1,d0
          rts
```

Here is the C function ParsePath(). Pass it a pointer to the path name you wish to decompose, and pointers to a 256-byte byte buffer for the directory portion of the path name, and a 31-byte buffer for the file name. The directory name, if there is one, will retain its trailing punctuation (':' or '/').

```
void ParsePath (path, dir, file)
char *path, char *dir, *file;
{
char *rindex();
register char *p;
```

```
    if ((p=rindex(path,'/')) || (p=rindex(path,':')))
       strncpy(dir, path, ++p - path);
    else
       p = path;

    strcpy(file, p);
    dir[p - path] = 0;
}
```

ConcatPath

This function accepts a pointer to a path name buffer containing a directory specification, along with a path node (directory or file) name to be appended to it. The node name is appended to the path name, and an intervening '/' is added if necessary.

```
void ConcatPath (path, file)
register char *path, *file;
{
int c;

   if (*path != 0 &&
         (c = path[strlen(path) - 1]) != ':' &&
         c != '/')
      strcat(path, "/");

   strcat(path, file);
}
```

NewExtension

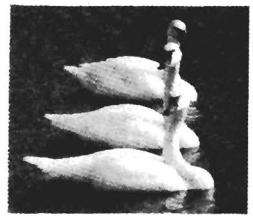
The third function in this trio accepts a path name and a file name extension, such as ".txt" or ".info". The final extension, if any, on the path name is replaced by the new extension. If the new extension doesn't begin with a '.', the function provides it.

```
void NewExtension (file, newext)
char *file, *newext;
{
register char *p, c;
short n;

   p = file + (n = strlen(file)) - 1;

   while (p >= file && (c = *p) != '.' &&
         c != '/' && c != ':')
      p--;

   if (p >= file && c == '.')
      *p = 0;
   else
      p = file + n;

   if (*newext != '.')
      *p++ = '.';

   strcpy(p, newext);
}
```


CygnusEd Professional

by CygnusSoft Software. Published by ASDG Inc.

"The New King of Editors,"
68000er, February 1988



"This editor is definitely one of the best I've used on any machine, and I've used many."
The Amiga Sentry, Sept. 1988

CygnusEd Professional brings unparalleled levels of flexibility and speed to the **Amiga.** features include:

Crash Recovery. Variable screen resolution up to 1000 by 800. Full Macro capability. Over 200 functions and variables available through AREXX. Hot Key operation. Screen refresh at over 30,000 cps. Search and replace at over 100,000 cps. Blitter based horizontal and vertical scrolling. Vertical and Horizontal block cut and paste. Many Word Processing features. Many Programmer features. Edit multiple files on screen at same time.

Asynchronous Printer support. Super interprocess communications facilities. DOS and AREXX commands can be launched from inside the editor. User definable "bookmarks". International keyboard and video support. Mouse or keyboard based Turbo Scrolling. Keyboard and mouse can be interchanged for everything except entering text. Will create icons for WorkBench users. Edits binary files. And **MUCH more!**

ASDG INCORPORATED 925 Stewart Street Madison, WI 53713 (608) 273-6585

Available At Your Local Dealer $99.95 U.S. Dollars

CygnusEd Professional Copyright 1988 By CygnusSoft Software. Amiga and Workbench are trademarks of Commodore-Amiga Inc. Developers are encouraged to contact ASDG for possible publication of your products

Schedule Assistant

Voice and Sound Reminders -
Create your own or use NAG PLUS library.

Perpetual Calendar *-Enter 198 events per day.*

Auto Dialer *-Connects YOU, not your modem.*

AREXX Port *-Commands any timed event or action.*

Notepad *-Click on any word to open text editor.*

Print • Search • And Much More.

Suggested Retail $79.95. Ask your dealer or contact:

NAG® PLUS 3.0

Gramma software For Your Own Good

Gramma Software
17730 15th Avenue N.E.
Suite 223
Seattle, Washington 98155
Phone #: (206) 363-6417
Fax #: (206) 361-0429

GFA-BASIC 3.0 for the Amiga

$139.95

GFA SYSTEMTECHNIK

Boldly goes where no BASIC has gone before.

- High-Speed Interpreter for easy program development
- Over 300 powerful commands
- FAST! – Execution times comparable to C
- In-line C and Assembler Commands
- Easy access to all Amiga libraries
- Extensive Amiga commands with submenus and built-in file requester
- Built-in Text Editor with syntax checking, procedure hiding and auto-indenting
- 400-page comprehensive manual
- Includes Run-Time Interpreter

Available NOW

from

Antic Software

544 Second Street, San Francisco, CA 94107

Call today to order:

800-234-7001

or see your local Amiga dealer.

(GF9200)

GFA-BASIC 3.0 is a trademark of GFA Systemtechnik, Germany; Antic Software and Antic are trademarks of Antic Publishing, Inc.; Amiga is a registered trademark of Commodore-Amiga Inc.

Reader Service Card

Fill out this card (please print) or attach your peel-off mailing label, and circle the reader service number for the ads that you'd like more information about (see Advertiser Index below).

Name _____

Company _____

Address _____

_____

Postal/Zip Code _____

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

TA V2I3

Advertiser	Page	Reader Service #
Antic Software	80	1
ASDG Inc.	79	2
CAPPCO	7	3
Central Coast Software	17	4
Checkpoint Technologies	49	5
CSA (Computer Systems Associates)	71	6
Electronics 2001	75	7
Fisher's Computers & Software	75	8
Forethought	71	9
Gramma Software	79	10
Intercomputing	Inside Back Cover	11
Intercore Development Systems	29	12
Interface Technologies	13	13
Lake Forest Logic Inc.	29	14
Lattice Inc.	Inside Front Cover	15
Memory And Storage Technology	1	16
Metadigm	62	17
Michtron	7	18
Newtek Inc.	55	19
OTG Software	71	20
Practical Solutions	49	21
Rainbow's Edge Productions	41	22
RAMCO	71	23
Safe Harbor Computer Supplies	62	24
The Puzzle Factory	55	25
Vidtech International Inc.	Outside Back Cover	26
Virtual Reality	7	27

for the AMIGA Transactor

The Magazine for Amiga Programmers

Please print your name and address below (or transfer your mailing label)

Check here for change of address ☐

Name: _____

Address: _____

City: _____ *Province/State:* _____

Zip/Postal: _____ *Phone:* _____ *CompuServe ID:* _____

NOTE: Prepayment required. Purchase orders will be accepted ONLY if accompanied by payment.

Subscribers only: please enter subscription number from mailing label ☐

Subscription Order Section

PLEASE NOTE THAT TRANSACTOR FOR THE AMIGA IS NOW PUBLISHED MONTHLY

	Canada ($Cdn.)			U.S.A. ($US)			Foreign ($US)			Air Mail ($US, overseas only)		
	Mag	Disk*	Both*	Mag	Disk	Both	Mag	Disk	Both	Mag	Disk	Both
12 issues	☐35.00	☐123.00	☐158.00	☐27.50	☐101.00	☐128.50	☐38.50	☐119.50	☐157.50	☐ 73.00	☐ 145.00	☐ 218.00
24 issues	☐65.00	☐220.00	☐270.00	☐49.00	☐160.00	☐190.00	☐69.00	☐220.00	☐290.00	☐135.00	☐ 250.00	☐ 390.00

* Ontario residents please add 8% of disk subscription price for provincial sales tax. No PST on magazine subscriptions.

☐ Please send Microfiche instead of magazines ☐ Send me info on Transactor Classic (C64/128)

Combo Subscription Bonus!

If you are subscribing to BOTH the magazine and the disk, check one of the following FREE gifts:

☐ Transactor Classic Amiga Disk #1 ☐ Fish Catalog Disk ☐ Extended Subscription (13 issues)

Product Order Section

Ontario residents please add 8% PST on all items

$Cdn.	$U.S.	
☐ 14.95	12.95	Transactor Classic Amiga Disk #1
☐ 14.95	12.95	Transactor Classic Amiga Disk #2

Fish Disks $6.95 US, $7.95 Cdn. each

Note: Disk numbers not yet released may be ordered in advance - simply add numbers to end of list. Disks 57, 80 and 88 are not available.

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12	☐ 13	☐ 14
☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24	☐ 25	☐ 26	☐ 27	☐ 28
☐ 29	☐ 30	☐ 31	☐ 32	☐ 33	☐ 34	☐ 35	☐ 36	☐ 37	☐ 38	☐ 39	☐ 40	☐ 41	☐ 42
☐ 43	☐ 44	☐ 45	☐ 46	☐ 47	☐ 48	☐ 49	☐ 50	☐ 51	☐ 52	☐ 53	☐ 54	☐ 55	☐ 56
☐ 58	☐ 59	☐ 60	☐ 61	☐ 62	☐ 63	☐ 64	☐ 65	☐ 66	☐ 67	☐ 68	☐ 69	☐ 70	☐ 71
☐ 72	☐ 73	☐ 74	☐ 75	☐ 76	☐ 77	☐ 78	☐ 79	☐ 81	☐ 82	☐ 83	☐ 84	☐ 85	☐ 86
☐ 87	☐ 89	☐ 90	☐ 91	☐ 92	☐ 93	☐ 94	☐ 95	☐ 96	☐ 97	☐ 98	☐ 99	☐ 100	☐ 101
☐ 102	☐ 103	☐ 104	☐ 105	☐ 106	☐ 107	☐ 108	☐ 109	☐ 110	☐ 111	☐ 112	☐ 113	☐ 114	☐ 115
☐ 116	☐ 117	☐ 118	☐ 119	☐ 120	☐ 121	☐ 122	☐ 123	☐ 124	☐ 125	☐ 126	☐ 127	☐ 128	☐ 129
☐ 130	☐ 131	☐ 133	☐ 134	☐ 135	☐ 136	☐ 137	☐ 138	☐ 139	☐ 140	☐ 141	☐ 142	☐ 143	☐ 144
☐ 145	☐ 146	☐ 147	☐ 148	☐ 149	☐ 150	☐ 151	☐ 152	☐ 153	☐ 154	☐ 155	☐ 156	☐ 157	☐ 158
☐ 159	☐ 160	☐ 161	☐ 162	☐ 163	☐ 164	☐ 165	☐ 166	☐ 167	☐ 168	☐ 169	☐ 170	☐ 171	☐ 172

The Transactor for the *AMIGA* Disk $12.95 US, $14.95 Cdn. each

(these disks correspond to issues of **Transactor** for the *AMIGA*)

☐ Vol. 1, Iss. 1 ☐ Vol. 1, Iss. 2 ☐ Vol. 1, Iss. 3 ☐ Vol. 1, Iss. 4 ☐ Vol. 1, Iss. 5 ☐ Vol. 1, Iss. 6

☐ Vol. 2, Iss. 1 ☐ Vol. 2, Iss. 2 ☐ Vol. 2, Iss. 3

Add flat $2.00 shipping & handling for any number of items ordered from above box.

Back Issues, Microfiche: $4.50 U.S., $4.50 Cdn. (MF=Microfiche, filled box=SOLD OUT)

☐V1#1 ☐MF ■V1#2 ☐MF ☐V1#3 ☐MF ☐V1#4 ☐MF ☐V1#5 ☐MF ☐V1#6 ☐MF

☐V2#1 ☐MF ☐V2#2 ☐MF ☐V2#3 ☐MF

Ontario residents please add 8% provincial sales tax for any items ordered from this box.

☐ Visa ☐ MasterCard ☐ Cheque/MO enclosed Cheque #_____ Dated ___/___/___ Amount _____

Acct. # _____ Expires ___/___ Signature _____

Please allow 6-8 weeks for delivery in the U.S., 4-6 weeks in Canada.

FOLD ALONG SCORES WITH CORRECT REPLY CARD ON OUTSIDE. USE TAPE, PLEASE - NO STAPLES OR GLUE.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 350 BUFFALO, NY
POSTAGE WILL BE PAID BY ADDRESSEE

Transactor
P.O. Box 338 Station 'C'
Buffalo, N.Y. 14209-9990

place stamp here

Transactor
10-85 West Wilmot Street
Richmond Hill, Ontario, Canada
L4B 1K7

Business Reply Mail
No Postage Stamp Necessary
If mailed in Canada

Postage will be paid by:

Transactor
501 Alden Road
P.O. Box 3250
Markham Industrial Park
Markham, Ontario
L3R 9Z9

Canada Post Postes Canada 46487

Bloopers

Volume 2, Issue 1 ***Shell Scripts Under 1.3:*** The script files that begin with comments will not work with the comments in place; remove them. Also, a necessary script file, used by the talking clock, was omitted. Here it is; it should be named *substr*

```
.key string,file/k,f1/k,l1/k,file1/k,f2/k,l2/k
.bra {
.ket }
.default file *
echo >{file} "{string}" first={f1} len={l1}
if {file1} not eq ""
echo >{file1} "{string}" first={f2} len={l2}
endif
; substring function
```

InterComputing Inc.

2100 N. Hwy 360, Suite 2101
Grand Prairie, TX 75050-1015

In business since 1984; Amiga support since 1985
Member of Dallas Better Business Bureau

We are a service oriented business offering discount prices

FAX: 1-214-660-3695

Customer Service and Order Status call: 1-214-988-3500

COMPLETE PRODUCT LISTING AVAILABLE

Call toll free: 1-800-622-9177

HARDWARE

MEMORY EXPANSION

Increase your Amiga's memory capacity and get more for your money!

FOR THE AMIGA 500:

A501 with 512k and clock	179.95
M501 with 512k and clock	179.95

FOR THE AMIGA 500 & 1000

Starboard II 0k	239.95
Starboard II 512k	Low!
Starboard II 1Meg	Low!
Starboard II 2Meg	Low!

FOR THE AMIGA 2000:

8 Up Memory Card 0k	159.95
8 Up Memory Card 2Meg	Low!
8 Up Memory Card 4Meg	Low!
8 Up Memory Card 6Meg	Low!

Drives

FData-10 ext. Drive	**139.95**
FData-20 ext. TwinDrive	**249.95**
int. 3.5" Drive A2000 only	145.95
AIR-Drive	169.95
Cal.Access CA-880	169.95
HardFrame SCSI Card	**249. 95**
Starboard SCSI Module	97.95
Quantum 80MB Drive	875.00
Supra 20MB HDisk	649.00
Supra 30MB HDisk	749.00

Sound / Midi

CMI Midi w/ Sync-Out	69.95
ECE-Midi w/ RS-232 thru	44.95
Future Sound Sampler	129.95
Soundscape Sampler	79.50
Perfect Sound A500/2000	64.95
Perfect Sound A1000	64.95
Midi Cable 5ft	8.95
Midi Cable 8ft color coded	11.95
Midi Gold for A2000 internal	69.95

Video / Graphics

A-Pro Draw 12x12	499.95
Amiga Live A 500	299.95
Amiga Live A 2000	329.95
Amiga Live A 1000	219.95
Color Splitter	85.95
Digi View 3.0 NTSC	139.95
Easyl 500	319.95
Easyl 1000	339.95
Easyl 2000	349.95
Flicker Fixer	499.95

Mitsubishi Multisyc Monitors for use with FlickerFixer :

13" Screen size	**549.95**
16" Screen size	**1349.95**
20" Screen size	**1895.95**

Panasonic 1410 Kit	249.95
Perfect Vision	175.00
Super Gen	699.95
X-Specs 3D	99.95

Miscellaneous

Mouse Pad	6.95
Multifunction Card	69.95
Phasar Gun	46.95
Sony 3.5" DS/DD Disks	18.95
Supra Modem 2400 baud with cable	149.95

SOFTWARE

NEW PRODUCTS

Animagic	**74.95**
Artistic ExpressionsVol. I & II, each	**39.95**
Battle Hawks 1942	**34.95**
Caligari	**189.95**
Gomf 3.0	**34.95**
Inovatools	**59.95**
Innvision	**169.95**
Kennedy Approach	**29.95**
Lords of the Rising Sun	**34.95**
Pro Video Font Set I/II, each	**94.95**

CALIGARI

Caligari Consumer *- A powerful home version of the Caligari's professional 3-D rendering software package* **199.95**

Business

BEST Management System	299.95
Cygnus ED	69.50
Excellence	179.95
MaxiPlan 500	99.95
Maxi Plan Plus	129.95
Microfiche Filer II	112.95
Money Mentor 2.0	59.95
Phasar 3.0	66.95
Prowrite 2.0	72.50
Scribble 2.0	57.95
Superbase Personal II	99.95
Superbase Professional	199.95
TexEd Plus	54.95
The Works! Platinum Edition	169.95
Wordperfect	197.95

New! New! New!

Extension Cables
for your disk drive or monitor
19.95 each

Desktop Publishing

Comic Setter	66.95
Font Set I	24.95
Laserscript	29.95
Newsletter Fonts	29.95
Professional Draw	139.95
Peofessional Page	239.95
Shakespeare	149.95
Studio Fonts	29.95
Zuma Fonts 1,2,3,4 each	24.95

Programming

A/C Basic Compiler	129.95
A/C Fortran (AB-Soft)	179.95
Aztec C Developer	199.95
Aztec C Professional	145.95
Aztec Source Level Debug.	59.95
Benchmark Modula-2	134.95
DSM Disassembler	42.95
GFA Basic 3.0	99.95
Lattice 5.0	219.95
Lattice-C ++	395.00
Lint	69.00
MCC-Pascal 2.0	79.95
Power Windows 2.5	56.9 5

Graphics & Video

Aegis Draw 2000	179.95
Animate 3D	99.95
Calligrapher	87.95
Comic Setter	66.95
Deluxe Paint 3.0	109.95
Director	44.95
Fantavision	39.95
Forms in Flight II	77.95
Interchange	29.95
Intro CAD 2.0	49.95
Lights, Camera, Action	52.95
Modeler 3D	72.50
Pageflipper Plus F/X	99.95
Photon Paint 2.0	109.95
Pixmate	49.95
Sculpt 3D	69.95
Sculpt 4D	399.95
Terrain Disk	19.95
TV Show NTSC/PAL	62.95
TV-Text	62.95
Turbo Silver	129.95
Video Effects 3D	129.95
Video Titler	99.95
Videoscape 3D	119.95
X-CAD	349.95
Zoetrope	99.95

Music

Audio Master II	69.95
Dr. T's DX-Heaven	99.95
Dr. T's Kawai K1 Libr.	99.95
Dr. T's KCS 1.6	166.95
Dr. T's KCS Level II	229.95
Dynamic Drums	47.95
Dynamic Studio	129.95
"M"	154.95
Midi Recording Studio	44.95
Music Mouse	49.95
Sound Quest Casio CZ	84.95
Sound Quest D-50	99.95
Sound Quest DX-7	99.95
Sound Quest DX-7 II	117.95
Sound Quest Generic	84.95
Sound Quest MT-32	99.95
Soundsc. Pro Midi Studio	124.95
Synthia	66.95
Texture	99.95

Utilities

Arexx	32.95
B.A.D.	32.95
Disk Mechanic	56.95
DOS-2-DOS	42.95
EZ-Backup	32.95
Face II	19.95
Project D	32.95

Flexible Data Systems

FDATA-20 Disk Drive

Fully compatible with your Amiga system, the FData 20 is very quiet and comes with an extra long cable. It is housed in a sturdy metal case and has it's own external power supply. Twice the storage of single drive units!

ONLY **249.95**

FDATA-10 Disk Drive

Still the best price & performance drive for the Amiga! **139.95**

Artistic Expressions

Clipart with personality!
Volumes I and II
Now Available!
Only 39.95

TRADE UP!

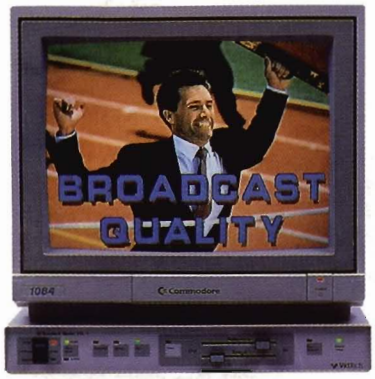
We will trade in your Amiga 500 or 1000 towards a new Amiga 2000 or 2500.
Call for details.

We also have a large selection of

GAMES

Write for our complete catalog.

As always we have the most 'customer friendly' terms: S/H $3.95 in cont. USA; Minimum order is $30.00; MASTERCARD & VISA with NO credit card fee; in Texas add 7% Sales Tax. We ship to APO addresses. RMA# required on all returns. 18% restocking charge. All prices subject to change without notice.



Coming To A Screen Near You.

If you're ready to become your own studio producer, there's only one desktop video system that will bring NTSC and S-VHS to your Amiga's screen. The Scanlock system by VidTech. Other systems give you a genlock that's "either-or." They either support NTSC or S-VHS. But not both.

Two For One

With Scanlock, you get two genlocks for the price of one. So you can migrate from NTSC to the newer S-VHS format without investing in another system. We also offer a Scanlock model that is compatible with the International PAL TV format, including its new S-VHS version.

Scanlock's dual-genlock system gives you broadcast-quality video at consumer prices. And our system offers all the bells and whistles, including front-panel fingertip controls, multiple monitor connections for viewing and editing, and independent fade controls for both reference and Amiga videos. If your Amiga is supporting a load of peripherals, we provide a connection that lets you use an optional external power supply.

Scanlock is a breeze to use. A "glitch-free" vertical interval switch lets you insert, reverse and remove reference video and Amiga graphics at the touch of a finger. And you can switch between NTSC, S-VHS and analog RGB formats without moving cable connectors. When you're done editing, you can resume normal Amiga operation with the flip of a switch.

Don't Get Locked Out

So if you're ready to become your own producer, choose the desktop video system that doesn't lock you into one genlock — the Scanlock system by VidTech. Coming to a screen near you.

Call or write us today: VidTech International, Inc., 2822 NW 79th Ave., Miami, Florida 33122. Telephone 800-727-2261 or 305-477-2228. Fax 305-591-1651. Prices:NTSC-$995, PAL-$1,095. Distributed by Southern Technologies, Dallas, Texas.

VidTech